ENGINEERING @ SP

SP Singapore Polytechnic

ET0104

EMBEDDED COMPUTER SYSTEMS

(Version 5.1)

School of Electrical & Electronic Engineering

ENGINEERING @ SP

The Singapore Polytechnic's Mission

As a polytechnic for all ages
we prepare our learners to be
life ready, work ready, world ready
for the transformation of Singapore

The Singapore Polytechnic's Vision

Inspired Learner. Serve with Mastery. Caring Community.

A caring community of inspired learners committed to serve with mastery.

The SP CORE Values

- Self-Discipline
- Personal Integrity
- Care & Concern
- Openness
- Responsibility
- Excellence

For any queries on the notes, please contact:

Name: Mr. Wee Boon Siong
Room: T12515
Email: wee_boon_siong@sp.edu.sg
Tel: 67721384

SP Singapore Polytechnic

SP SCHOOL OF Electrical & Electronic Engineering

Industry-Collaborated TECH HUB

Diploma in Aerospace Electronics

Aerospace Engineering Hub

Themed Project : Unmanned Aircraft Applications
Facilities : A320 Flight Simulator
Industries : Aerospace, Air Transport
Industry Partners : AIRBUS, Rolls-Royce

Industry Co-location Lab : ST Engineering Aerospace

Diploma in Computer Engineering

5G & AIoT Technology Hub

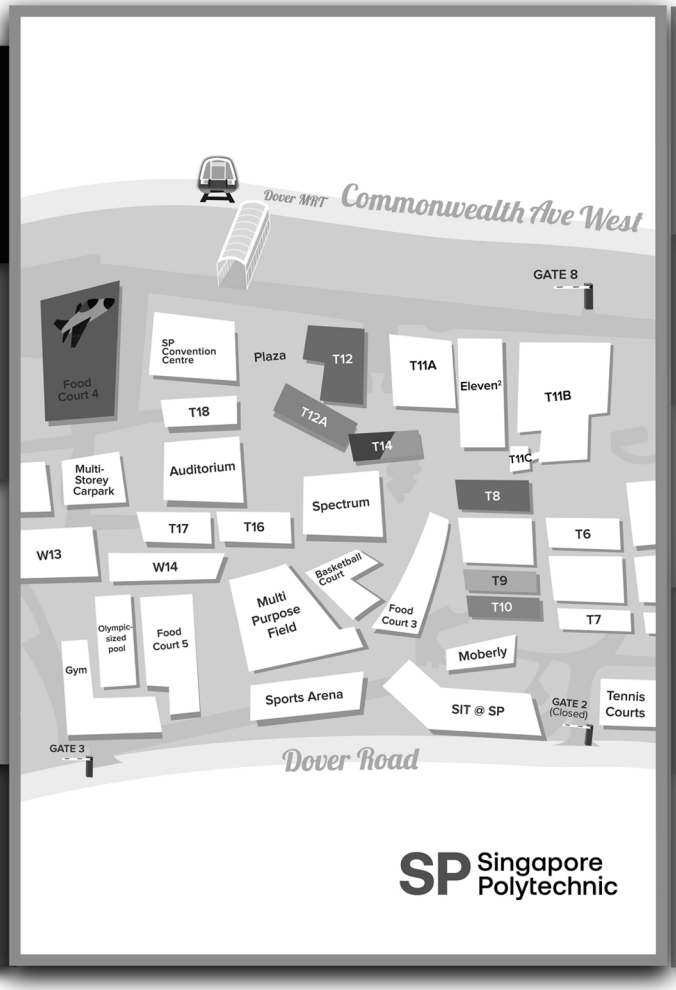
Themed Project : Development of 5G & AIoT Use Cases
Facilities : 5G Garage, Singapore 5G & Telecom Academy, 5G & AIoT Centre, SP Smart Connected Solution Lab
Industries : ICT & Media, Smart City
Industry Partners : ADVANTECH, aws, climateasia, ENVISION digital, GOVTECH, HUAWEI, mVIZN, vmware, DELL

Industry Co-location Lab : BOSCH Invented for life, ERICSSON, Singtel

Diploma in Engineering with Business

Technology to Business Hub

Themed Project : Technology To Business Venture
Industries : Entrepreneurship & Start-up
Industry Partners : APOL, BevEat, GEX VENTURES, OCTOPUS8, Packsdot, SMARTDB, SPGG, Vaster



Diploma in Electrical & Electronic Engineering

Robotics, Automation & Control Hub

Themed Project : Smart Workflow - Secured Supply Chain Solutions
Autonomous Mobile Robot
Facilities : SP-SICK Smart Sensor Lab, SP-Rexroth I4.0 Innovation Lab, SP-Siemens Digital Experience Centre
Industries : Logistics, Electronics (Manufacturing), Electronics & Chemicals (Manufacturing)
Industry Partners : ABB, BECKHOFF, KUKA, SEW EURODRIVE, Weston Robot, ZEBRA

Industry Co-location Lab : rexroth A Bosch Company, SICK Sensor Intelligence., SIEMENS

Power & Sustainable Energy Hub

Themed Project : Energy Efficient Eco Organic Agritech Testbed
Facilities : Low Voltage & High Voltage Training, LE & LET Training, Energy Efficiency Lab
Industries : Energy & Power
Industry Partners : VST, SIT, SPGG, SPgroup

Industry Co-location Lab : CHNT, sembcorp

Semicon & Electronics Hub

Themed Project : Artificial Intelligence & IoT Using FPGA/Embedded Solutions
Facilities : SP-Advantest Test Engineering Centre, Nanofab & IC Design Labs, Infineon Innovation Lab
Industries : Semiconductor & Electronics
Industry Partners : SSIA

Industry Co-location Lab : ADVANTEST, Infineon

Biomedical Engineering Hub

Themed Project : Engaging Rehabilitation with Data Visualisation/ Analytics and Applications of Wearable Devices
Facilities : Biomedical Engineering Lab
Industries : Healthcare
Industry Partners : GE Healthcare, GOLD LITE, Khoo Teck Puat Hospital, KK Women's and Children's Hospital, NATIONAL INSTRUMENTS, SENZE HUB, SingHealth

Autonomous & Transportation Systems Hub

Themed Project : 5G-enabled Autonomous Safety Vehicle
Autonomous Underwater Vehicle
Facilities : RSS Lab for RTT Training, SP-TIBCO Adv Mfg Analytics Lab
Industries : Smart Transport, Land Transport
Industry Partners : AOS OFFSHORE, Building and Construction Authority, JM VisTec System, Land Transport Authority, SBS Transit, SIIX, SMRT, ST Engineering Land Systems

Industry Co-location Lab : TIBCO

CONTENTS

Module Overview

1 Introduction

Embedded Microprocessor Systems is a third year module for the Diploma in Computer Engineering (DCPE) course.

Students must have good understanding of microcontroller technology and programming in a high level language, preferably "C".

2 Module Aims

To provide students with knowledge and understanding of low cost and small size but powerful embedded computer systems, used commonly for industrial and home devices with graphical interfaces. The practical use of such systems is shown by how to design and interface to such devices.

1 Introduction to Embedded Systems

1.1 Introduction

Embedded computer systems are special versions of the computers we use everyday. The word "embedded" means part of another system. We will call this the host system. Indeed, much of the intelligence of today's devices owes much to the computers which are part of them. These range from devices like our remote controls, to washing machines, automobiles and aircraft. These may have several embedded systems in them. Embedded systems are typically smaller in size as compared to desktops. A quick review of microcomputer systems is in order.

1.2 Typical Microcomputer Organisation

A basic computer system consists of the Central Processing Unit (CPU), the memory unit and the input and output units. The units are interconnected with buses.

1.2.1 Central Processing Unit

The central processing unit is made up of three parts : the arithmetic/logic unit (ALU), registers and a control unit. The ALU performs arithmetic and logical operations on binary data while the registers are used for storing data, addresses or instructions. The exact number and type of registers depends on the particular computer system. The main register is called an accumulator which usually stores data to be manipulated by the ALU at the beginning of an arithmetic or logic operation. At the end of the operation, the accumulator usually stores the result.

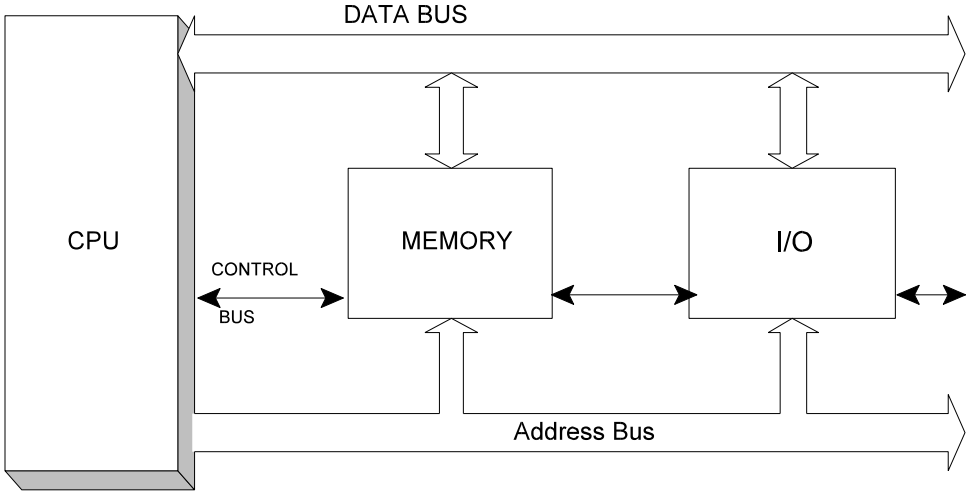


Figure 1 Block Diagram of a Basic Computer System

Control Unit

The control unit directs the operation of all other unit, by providing timing and control signals. This unit contains logic and timing circuits that generate the proper signals necessary to execute each instruction in a program. The control unit is also capable of responding to external signals; for example, interrupts. An interrupt request will cause the control circuit to temporarily interrupt the main program execution, jump to a special routine to service the interrupting device, then automatically return to the main program.

Memory Unit

The memory unit contains a large number of memory cells, usually RAM and ROM. They are used to store instruction sequences or programs which the computer will execute, the data that is to be processed by the programs, as well as data resulting from the processing. Operation of the memory is controlled by the control unit which signals for either a READ or a WRITE operation.

Input/Output Units

Input devices are used to supply the data needed for processing or to tell the program how to operate on data. Output devices allow results of the computer operations to be supplied to the user. The input/output devices are also known as peripherals. Common input devices are keyboards and mouse. Examples of output devices are printers and visual display units.

Input and output ports are simply the interface circuits which provide the appropriate electrical connections between the input/output devices and the rest of the computer system. Physically an input or output port is just a parallel set of buffers or D flip-flops which latch data when strobed by the CPU.

1.2.2 Bus Structure

The CPU is linked to other parts of the system by buses. A bus is a group of conductors that has connections to all of the system's main blocks. There are three major buses: the address bus, data bus and the control bus

Address Bus

The address bus is used by the CPU to send out the address of a memory location that is to be read from or written into; or the address of a particular input or output port. The address bus is unidirectional. Information generally moves in one direction only, originating in the CPU only.

Data Bus

The data bus is bidirectional because data can flow to or from the CPU, memory, or the I/O ports. Any device outputs connected onto the data bus must have three-state buffers, so that they can be floated except when that device is being selected. The number of conductors in the data bus is determined by the number of binary digits the CPU can handle at a given time. However,

the quantity of data stored at each location depends on the particular memory device. It may be a single bit, or 4, 8 or 16 bits per location.

Control Bus

The control bus is used to carry signals which synchronise the activities of the separate blocks of the computer system. It tells the memory and the I/O ports whether the CPU is reading or writing data. Two conductors found in the control bus are the READ and the WRITE lines. The READ line activated by the CPU when the CPU wants to retrieve data from the memory or input data from an input device. For example, to read a memory location the CPU places the address on the address bus and a READ signal is placed on the control bus. The memory then outputs the data from the addressed location to the data bus which is then transferred to the CPU data register. When the CPU is to store in memory or output data to an output device, it places the data on the data bus and then activates the WRITE line.

There are also some other control lines present in CPU to perform more complex operations.

1.3 History of microprocessors

Electronic computers were developed in the 1960's.They were physically large systems which occupied a room. In time, the size became smaller, until it was able to fit into an integrated circuit the size of a thumb. Intel Corporation designed the 4004 microprocessor in 1978, for a calculator. This was revolutionary, as previously computers were designed on separate large printed circuit boards and connected together.

The following lists the developments in terms of the various microprocessors from Intel. But it is representative of the progress in computing power for the industry.

	8080	8085	8086	8088	80286	80386	80486	Pentium	Core ix
Address	16	16	20	20	24	32	32	32	64
Data	8	8	16	8	16	32	32	32	64
Maths	N	N	N	N	N	Y	Y	Y	Y
MHz	1	3	5	5	16	25	133	3k+	3k+

As processors progress in terms of performance, older models get obsoleted. However, the technologies get passed on in embedded systems. Thus we can expect embedded systems to become more powerful as time goes on.

A major challenge computer manufacturers face is the ever increasing demand for more computing power as users find more uses for computers. But users should not be expected to discard their old programs immediately to take advantage of the changes. Program development is a significant investment of effort.

1.3.1 The need for more computing power

As more uses are found for computing power, greater demand is found for its use. One area is the use of image and video data and analytics. An image can easily occupy one megabyte of memory. This requires a large addressing space. Also, manipulating these data require the use of fractional or floating point numbers, rounding off is not an option here.

Furthermore, fixed and floating point numbers need a large number of bits to represent data. The IEEE format specifies 80 bits. Many processors have incorporate hardware mathematical coprocessors to take advantage of this.

Sophisticated control algorithms like fuzzy logic, neural networks and statistical signal processing use a large amount of floating point numbers. Often they use matrix operations as well.

So, while embedded systems may control simple devices, the mathematical background required to do this *optimally* and *accurately* requires a lot of computing power, not just for games on desktops!

1.3.2 Increases in performance

This can come about in several ways:

Processing speed

Changing the processor speed brings about immediate benefits in terms of faster processing. The benefits are that older programs can be used. But this is not enough as computer programs deal with more complex data.

Address

Manipulating data like sound, images and video need large storage and high speed. These data need to be placed in memory for efficient processing. A 16 bit address only allows 2^{16} or 65536 bytes of data to be accessed. Today's processors use 32 addresses, with 64 bits are becoming available. It is a major challenge to modify programs to take care of this change.

Data

Most of the time, a unit of data manipulated is 8 bits or a byte. More sophisticated mathematical manipulations require many more bits to prevent loss of precision. Depending on whether one is using fixed or floating point, using 80 bits for mathematical data is common.

Also, instead of fetching 8 bits at a time, it is more efficient to fetch *and* manipulate them in units of 32 bits at a time, hence the reference to 32 bit data. Thus newer processors perform additions, etc on 32 bit data!

This can also cause problems in high level languages as the definition of an integer (the basic unit of data manipulation) will change when moving to a new processor.

Architecture

This refers to changing how data flows within the processor, making it more efficient. For example, having two or more processing units allow simultaneous fetching of data and actual processing of data.

Also we may pre-fetch data or code that is not required yet, anticipating when it is used and place it in memory that is faster so it executes faster.

The hardware may also be optimized so that certain commonly instructions can be executed much faster than others.

Architectural changes may introduce subtle errors in programs if they are time dependent. Otherwise, it should not change the running of the program significantly.

1.3.3 Progression of Intel microprocessors

The Intel series of microprocessors became popular because of the IBM PC which was introduced in 1982. It is remarkable that programs used then can still be run on today's desktop systems. This is not true for the 8080 and 8085 which are compatible with each other, being 8/16 data/address bit processors. As a historical note, the 8088 was designed to help software developers bring their programs over from the 8085. This was done using a segmented address scheme, allowing a 16 bit "logical" address within a 20 bit address space. The 20 bit address quickly became inadequate, prompting new address schemes involving 24 and 32 bits. As mentioned earlier, it was important that old programs could still be run.

In order to provide compatibility, Intel allowed the later processors to run in Real, Protected and virtual 86 modes to run programs written for earlier processors. See the chapter appendix for more information.

1.3.4 Advantages of the Intel PC architecture

While this section is not an advertisement, it serves to highlight what benefits processor hardware developments have on the industry.

The use of the PC over the decades have brought about many thousands of hardware and software applications. Not to mention the books and courses that build on this architecture. This translates

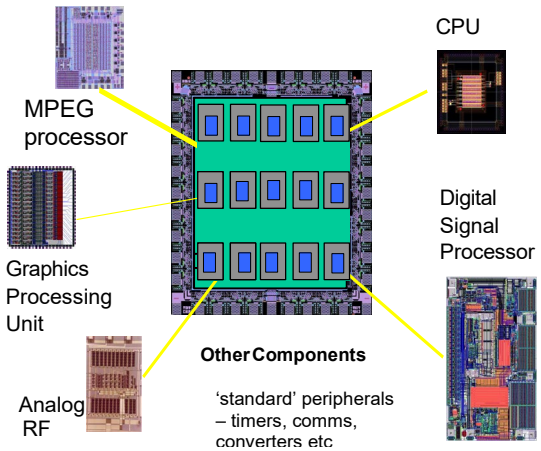
into quick application development for sophisticated applications. Also, the processing power of the Intel processors continue to increase.

The Intel line up of processors represents the mainstream progress of desktop computing and thence to embedded systems. Meanwhile another family of processors have been gaining widespread use due to their use in mobile devices. The ARM (Advanced RISC Machine) processor was introduced in 1983. The term RISC (reduced instruction set computing) was a design philosophy that emphasized simple machine instructions, large number of on-chip registers and the use of only load and store instructions to interact with memory. These features allow for low power consumption and hence, their popularity in mobile devices.

The ARM processors are designed by their parent company who license the design to other companies who customize the design by adding in other hardware and peripherals. These companies then manufacture their version of the chip, unlike Intel who do all their design and manufacture in-house. Examples are: Qualcomm (Snapdragon), Mediatek, Samsung (Exynos), Apple (M1). Due to their high level of integration and complexity, these processors are often referred to as SoC (systems on a chip).

The peripherals can include hardware like GPUs (graphics processing unit), analog radio frequency processing and so on.

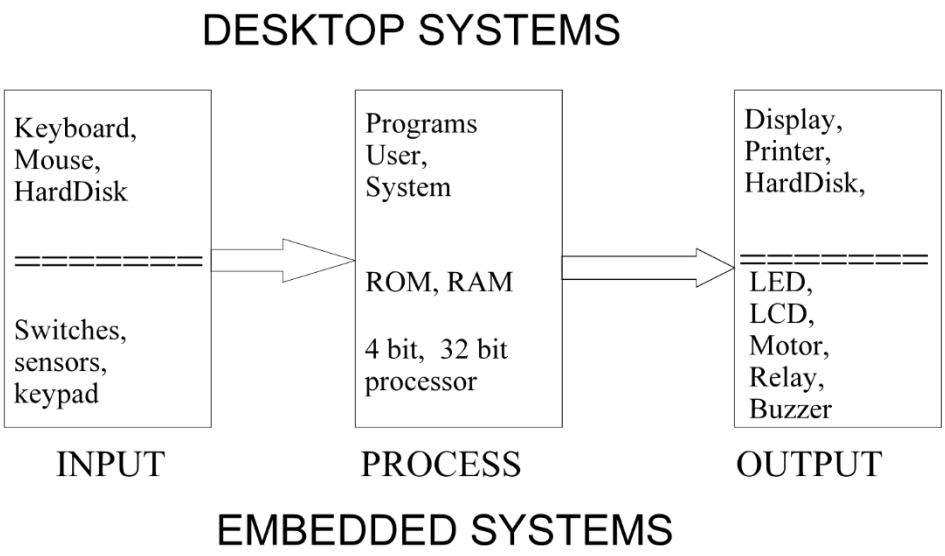
A sample SoC is shown in the adjoining figure.



We will examine in some detail, the version made by Broadcom, as this is the CPU used in the Raspberry Pi series of processors.

1.4 Embedded Systems Characteristics

It is useful to compare the characteristics of the commonly used desktops with embedded systems.



As can be seen, the inputs to a desktop system differ from those of an embedded system. The processors however, may be similar. The amount of memory will differ, desktops being able to physically access more amounts of memory. Again, the outputs are different.

Fixed use

These systems are used for a fixed purpose. This is normally to control another device or process. Desktop computers can be used for a variety of purposes - games, word processing and so on.

Small size

The system must be relatively smaller than the system it is part of. This places constraints on the physical size of the embedded system.

Limited resources

This is related to the point about physical size. The amount of memory is limited and most of the time, this is in terms of RAM. Embedded systems mostly never use hard disks and programs are stored in ROM.

Failure tolerance

Because of the unpredictable nature of the physical environment, the systems should be capable of failing "gracefully" if it meets unpredictable circumstances. This is much preferable to the system "hanging".

Real time control

The speed of the embedded system is fast enough to affect the operation of the environment. For example, a remote control device operates in terms of seconds. But an aircraft control system works in terms of thousandths of a second. Of course the complexity of the calculations required for such a system plays a major part in determining what kind of system to use.

In many cases, especially involving human interactions which take place in fractions of a second, it does not make sense for a computer to work in millionths of a second here.

Guaranteed response time

Closely related to real time control, is the need for the embedded system to respond to a device or user request in a minimum time. This is especially true when systems are running several tasks at the same time. Important processes have stringent conditions on this.

Reliability

An unusually high emphasis is placed on this, especially when human lives are concerned. To complicate this is the fact that embedded systems often work in environmentally hostile environments. Aircraft are subjected to extremes of heat and cold and mechanical vibration and shock. Other factors are electromagnetic interference, radiation and chemical pollutants. Whatever environment the host system is operating in, will subject the embedded system to the same.

Being part of a larger system, the embedded system may be installed in physically inaccessible places. It may be very difficult to perform software updates and have to run continuously for years on end.

Simple input and output

Since embedded systems help control a larger system, it does not need elaborate input devices like a keyboard or a mouse or a display screen or a sound system for output. Most, it will read from switches, sensors, or small size keypads. It may output to motors, relays, buzzers, light emitted diodes, liquid crystal displays. Sometimes, a touch sensitive screen is used for input and output.

Power

Being part of a larger system, an embedded system should not consume too much power relative to the host system.

Low cost

This is needed, so it does not add appreciably to the total cost of the host system.

1.5 Some hardware features of embedded systems

Embedded systems frequently have to work with devices external to itself. For example, it has to receive input, from a keyboard, move an object and display the results of its processing to a display device like an LED display.

Mechanical control has to be in a precise manner. As mentioned before, the system often has to service several devices at the same time. Some of these devices have a large amount of data to be transferred in short time. Various hardware techniques have been built into the processor to help perform these tasks. These are: i) interrupts ii) hardware timers iii) direct memory access.

However, an emerging trend is to increasingly incorporate embedded systems into everyday life, such as smartphones, fitness trackers and so on. The issue of power management becomes very important here.

1.5.1 Interrupts

Embedded systems work at a high speed, in terms of millions of instructions per second, but often it interfaces to devices which operates much slower than it. Data input from humans for example, occur at millisecond rates, and the eye cannot detect events occurring at speeds faster than this. For the processor to wait for input and output is time consuming and inefficient. One way to improve this situation is through the use of interrupts.

1.5.2 Timers

In order to control external devices in a precise manner, it is important to have a source of precise time events. While software delay loops may work and are simple, there are problems:

i) These normally involve loops with a high count. This does not allow the processor to perform any other tasks.

ii) In modern processors, the execution time of an instruction can vary, depending on various instruction pre-fetch schemes, like caching.

iii) A proper multitasking operating system cannot work on software loops as this makes task management very difficult.

Thus timers, external to the CPU, but which may be *on* or *off*-chip are essential. Timers may be used in many ways, but the most common is to:

i) Set a value to the timer
ii) Allow it to count down, and then cause an interrupt.
iii) The frequency of interrupts can be set so that a user can call up a delay using convenient units like milliseconds.

It is possible to directly read the values of the timer register for short duration events. As mentioned before, timer interrupts are the basis of multitasking operating systems.

1.5.3 Direct Memory Access

Direct Memory Access (DMA) is a method where peripheral devices transfer data to and from memory *without* data going through the processor. This is faster than using software. This is described in the chapter appendix.

1.5.4 Power management

The availability of low cost and low powered embedded processors led to the widespread use of the smartphone. Now the data transferred over today's mobile phones are heavily compressed digital data, and not analogue signal. Thus the networks that enable the widespread use of such phones are well equipped for handling data. This is served by the concept of the cellular network, which uses large numbers of base stations which transfer data between them and have large numbers of wireless end points which allow device with low transmission power to communicate with the base stations and from there, the larger network of connected based stations.

Concurrently, the use the Internet has expanded from proprietary, scientific and military data sharing networks to encompass a very wide range of applications such as business, commerce and personal mail. In the 1990's there were some efforts to equip consumer devices with the ability to be controlled remotely and also to report their status over the Internet.

These two factors have joined together to give us the current state of connectivity, the Internet of Things (IoT). In today's data driven computer applications, data is being "mined" to obtain information about trends, anomalies to better predict and thus plan for the future. A device using IoT technology can collect how much personal information an simply from the user's interaction with it. A thermostat, for instance, knows your sleep/wake cycles and when your home is empty. Wearable products can learn a lot about your health, movements, location, and activity levels. Connected cars know where you go and how well you drive. And by correlating multiple information sources, it is possible to infer a great deal about your habits, preferences, and activities. From the consulting firm TechTarget: *The Internet of Things (IoT) is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction.*

However a large proportion of embedded devices run on their own power and thus there is a great need to conserve this resource. This is emphasised with the world wide realisation of the earth's dwindling energy resources and of global warming due to energy production. This is thus an important area of development for embedded computer systems. A recent study showed how power savings can be achieved in such systems.

We will provide an overview of current methods of power saving design in 3 areas of embedded system design: program, power, processor

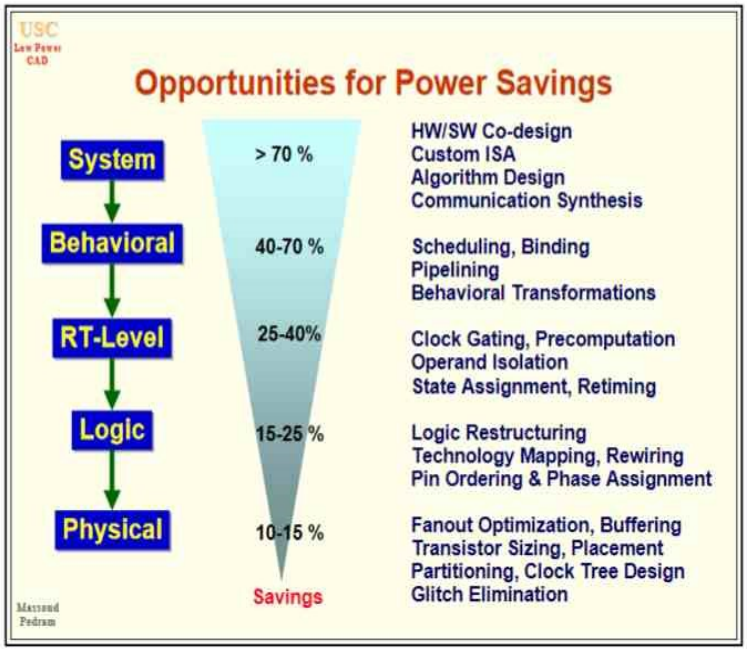


Figure 3 Power savings in embedded systems (Massoud Pedram)

Program Design

Since energy used by the system is proportional to the number of instructions executed, it makes sense to use the least possible number of instructions that can accomplish a task. However, it should be noted that high level languages such as C actually generate several lines of assembly code and thus a good knowledge of the processor architecture is necessary.

Switch to low power modes as soon as possible. The CPU is turned off but the contents of the working memory are preserved according to the mode:

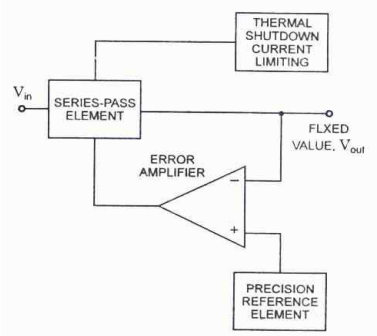
Standby - contents of memory are still present, using battery power.
Hibernate - contents of memory are saved to nonvolatile memory, like flash or magnetic storage

In the context of interrupts, the main task now consists of a "standby" command and a timer or peripheral interrupt will wake up the processor for processing data, which can be done at high speed. The speed at which the processor can restore it's working state is important and will decide the low power modes to use.

As shown in Fig 3, greater power savings can result if an algorithm can be implemented without using a processor and RAM, which takes up the most power. Field programmable gate array (FPGA) or ROM based code in a separate piece of hardware would use very little power.

Power Supply

Current electronic devices need a stable voltage to operate. The standard 7805 voltage regulator integrated circuit has been used for decades to supply a stable 5 volt power supply to logic circuits that operate at TTL logic levels.

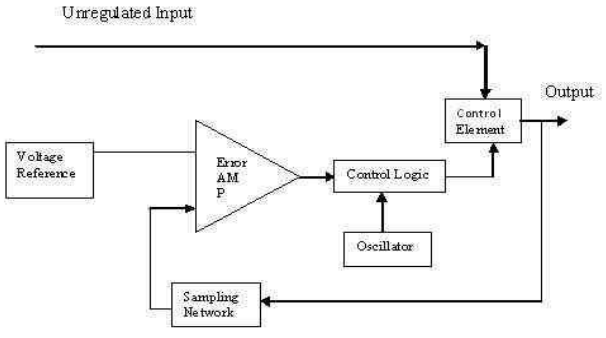


Fundamental Block Diagram of a Three-Terminal IC Voltage Regulator

An often ignored waste of power occurs due to the dropout or forward voltage of a device. This is the voltage required for a device to be in an operating mode. For example, for a normal diode this is around 0.6 volts and larger than 0.2 volts for a transistor. In the design of the 7805, a dropout voltage of 2.5 volts is needed to keep it in operation. If a current of 1 ampere is passed through, a dissipation of 2.5 watts is needed which will necessitate a heat sink to be attached to the device.

This is clearly a waste of power and there is now, low drop out (LDO) voltage regulators are used. These need a dropout voltage of only 0.6 volts to operate properly.

However, the input voltage to the LDO must be supplied. It is difficult to always find a transformer with the needed voltage and furthermore, this voltage can vary with the load. Another way of supplying this voltage is through the use of the switching regulator. Basically the incoming DC voltage is converted to a pulse wave through an oscillator and an inductor used to step the voltage up (boost) or down (buck). The device measures the difference between the voltage reference and the output voltage and changes the duty cycle of the pulse or its frequency to maintain the output voltage. Up to 90% efficiency can be achieved with this method. However, this adds cost and complexity together with a higher noise level and poorer regulation.



Block Diagram of Switched Mode Power Supply

There are a wide variety of batteries available and the technology changing. There are also primary (nonchargeable) and secondary (chargeable) technologies and various charging techniques.

However it is also possible to replace batteries with energy harvesting as there are sources of energy surrounding us. There are natural sources such as sunlight, wind and water, and those generated by body motion such as footsteps and hand swings.
Also there are those generated by daily use, such as radio frequency signals from cellular transmissions, radio, television and electromagnetic signals leaking from transformers.

Processor hardware
Standard 5 volts were used to power integrated circuits and embedded systems in the past. Today, 3.3 volts are very common and some devices can operate as low as 1.8 volts.

The rate at which a given piece of software executes is one factor that affects the power consumption. The lower the frequency of operation, the lower the power consumption. For example, imagine that a CPU needs to execute 100,000 instructions of some software to get a job done and this needs to be performed every second. If the CPU were running at a clock frequency that enabled it to execute a million instructions per second, it would be capable of doing 10X the amount of required work. So, lowering the frequency of the clock by this amount [to facilitate 100K instructions per second] matches performance to requirements and optimizes power consumption. A higher voltage is needed for operation at higher speeds.

Appendix

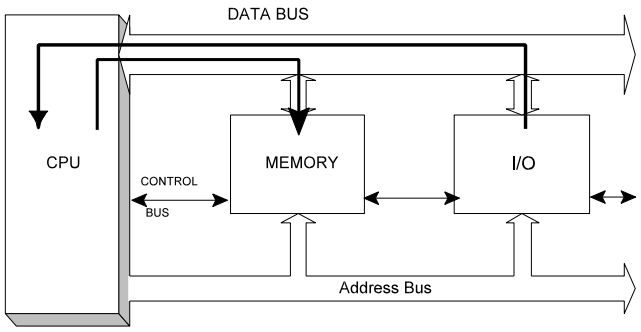
Real, Protected and virtual 86 modes

In Intel terminology, this involved starting processors in *real* mode, compatible with the 8088 to run old programs. If desired, they could switch the processor so it operated in *protected* or full 32/32 data address bits. In addition there was also a “virtual 86" mode which allowed *multiple* 8088 programs to run in protected mode.

DIRECT MEMORY ACCESS (DMA)

Software operations doing just a read and write can take many processor cycles. However, a DMA controller circuit generates the necessary address and control signals and can transfer data on every clock cycle. The main processor goes into a *HOLD* state where all its address and data lines are in high impedance.

Let us consider the case where a device is transferring data to memory, that is, we are doing a read. The case for writing to a device is the same, except for the direction of data transfer and of course, the control signals.

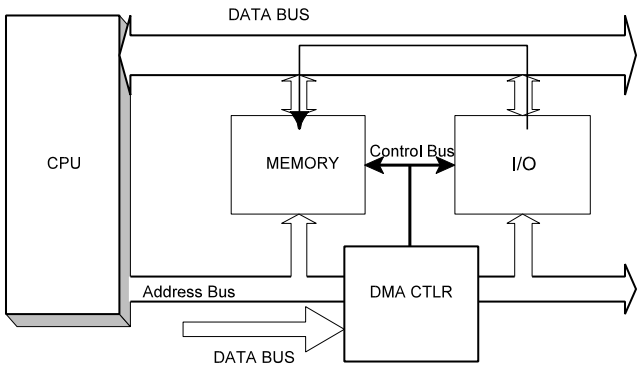


Without DMA

In this instance, the CPU reads from the I/O, and then writes to memory. All data passes through the CPU. Furthermore, there is also a software loop with a counter to keep track of the number of bytes transferred. So the instructions required are:

i) Read from I/O
ii) Write to memory
iii) Increment counter
iv) Check for end of loop
v) Loop

With DMA



A DMA controller will generate the addresses needed and keep track of the bytes transferred, using hardware, which is much faster. The CPU is effectively inactive, its address and data bus is taken over by the DMA controller. Note that the DMA controller is treated as an I/O device as well and connected to the data bus for initialization and status purposes only.

2 Bus Systems and Devices

2.1 Introduction

Many of today's advanced embedded processors have the hundreds of kilobyte sized on-chip memories of desktop computers just a few decades ago. They also run at gigahertz speeds and have a high level of integration with other functional blocks on the same chip or in what is a System on Chip or SoC. These memories are used for programs and data executing within the processor and are optimized for speed so that the busses in the processor are normally not easily accessed.

However these processors need to interface to external devices which have grown in complexity. A common situation is to acquire and store data or interface with large displays. An external bus interface is required for such situations. So this external bus interface is used for i) specialized memory needs, for example, static RAM, NAND and NOR type flash memory ii) controlling a large graphics LCD.

Thus many vendors provide an external bus interface and most of the time, a high speed bus is not needed. As we will see, just using a few low cost TTL chips are enough to do the interfacing. However, the protocols used in these simple bus interfaces still lives on in modern systems, which build upon these protocols with advanced techniques in computer architecture. We will look at a bus and see how it interfaces with the most common kinds of devices, namely memory chips, buffers and latches.

2.2 ARM based Hardware architecture

We have seen that the ARM series of processors has become dominant in mobile computing and most of these processors are used in a specific application. However there have been versions of the ARM that have a more open interface. One of these is the Raspberry Pi (RPi), a low-cost single-board computer with a SoC from Broadcom - this SoC is widely used in set top boxes used in televisions.

The RPi is the size of a credit card and can run the Linux operating system. Its hardware can interface with low-level electronics and so that high-level Linux software can analyse as well as provide control to external devices. Thus it has been successfully used in Internet of Things (IoT) applications, as well as robotics, cyber-physical systems, 3D printing and much more. Besides the low cost, the success of the RPi as an affordable computing environment is due to the effort made by the producers of the RPi to make its operating software trouble-free and easy to use especially for new users.

For industrial use, the designers came up with the Compute Module 3 (CM3) which has much less built in peripherals than the RPi. Much more processor pins are available for use, lowers the cost of the board making it more flexible for custom designs.

2.3 Hardware of the SoC

The Broadcom SoC used in the CM3 has a CPU which uses an ARM Cortex A core which follows the ARMv8-A architecture. The following is a list of hardware blocks which typically make up the SoC.

- GPU (Graphics Processing Unit - Broadcom VideoCore IV)
- Memory
- Timers
- DMA (Direct Memory Access)
- Interrupt Controller
- GPIO (General Purpose Input Output)
- USB (Universal Serial Bus)
- PCM (Pulse Code Modulation)
- I2S (Inter-IC Sound)
- PWM (Pulse Width Modulation)
- Serial Communication
  - I2C (Inter-Integrated Circuit)
  - SPI (Serial Peripheral Interface)
  - UART (Universal Asynchronous Receiver/Transmitter)

On the Broadcom SoC, an on-chip bus transfers data between these various blocks using hardware based on a bus architecture designed by ARM called AMBA (Advanced Microcontroller Bus Architecture). This in turn specifies other interfaces to external devices like the Advanced eXtensible Interface (AXI) which allows interfacing to external memory devices. There are other on-chip interfaces in use by other companies.

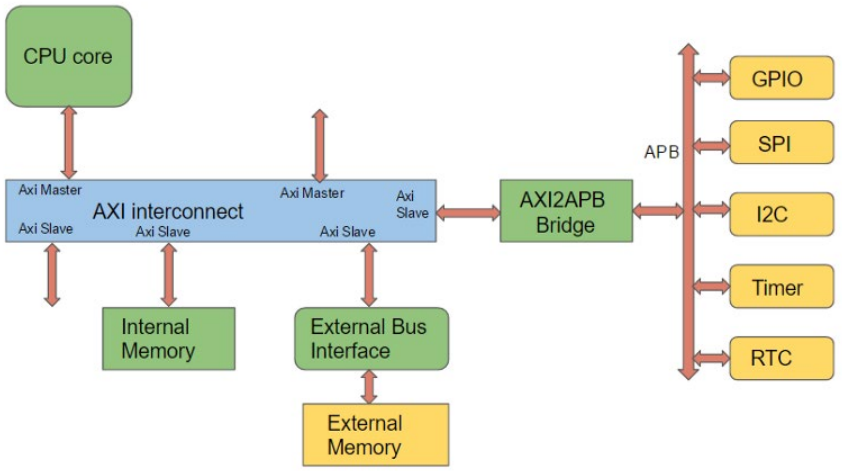


Fig 2.1 ARM AMBA block diagram

The AXI specification allows connection to external memory using the external bus interface and we will look at how it can be used to access flash memory using the NAND type. For the Broadcom devices, this is known as the Secondary Memory

Interface or SMI. The NAND Flash memory interface requires a smaller number of address pins and six are specified in the SMI with the requisite control pins. The "8080" or "6800" bus interface is used because of their simplicity and modest hardware requirements.

The 8080 from Intel and 6800 from Motorola were among the very first microprocessors on the market in the 1970's. These devices are not in use today, but their interfacing protocol is still very much in use and this motivates for learning about how they work.

The 8080 was electrically cumbersome to use due to its multiple voltages and support chips and an improved version, the 8085 was introduced soon. In fact it is the bus timing signals of the 8085 which is commonly referred to as the “8080 bus” in current use. Currently, there are several variations on this specification. In this case, we will describe how the *8080 mode* parallel bus works.

2.5 The 8080 mode bus

In the earliest commercial microcomputer systems, vendors have always provided an interface bus. They realized that it was not possible to anticipate all the needs of users who needed to interface to all kinds of devices. The 8080 interface specifies a 16 bit address bus, an 8 bit data bus and the use of separate read and write signals for control.

Current devices that interface using the 8080 mode are graphical liquid crystal displays (LCD) and various types of flash memories. In these devices the address, data bus sizes can vary but the timing of control signals have a common sequence.

2.5.1 Description of bus

The 8080 and 8085 originally specified the following bus widths and control signals.

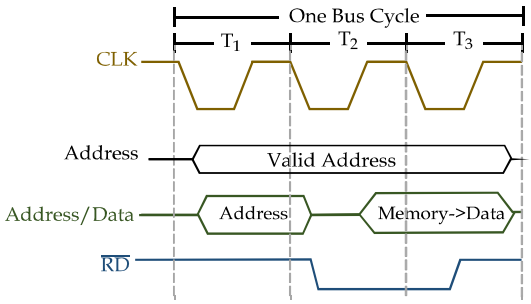
Table 2.1 Description of selected signals of the 8080 mode Bus

Signal name	Description
A0-A15	This is the address bus and gives a total of 64 Kb of memory. When outputting to an I/O device, only 8 bits are used.
D0-7	These are the data bus bits for devices, signifying that a byte is the normal size of data being accessed.
$\overline{\text{RD}}$	*Read* instructs a selected device to drive data onto the data bus.
$\overline{\text{WR}}$	*Write* instructs a selected device to store the data on the data bus.

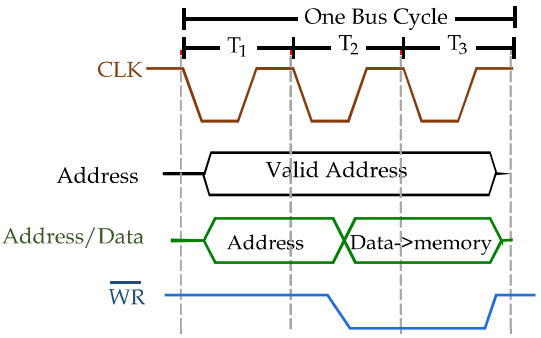
$IO/\overline{M}$	*IO/M* indicates whether the current instruction is mean for memory or I/O devices.

As mentioned earlier, for devices in current use, the full width of the address and data busses can vary. For example, the data bus may be 8, 9 or 16 and 18 bits.

The timing of a typical instruction of the 8080 and 8085 is that it executes over 3 clock cycles. Also the data bus is multiplexed allowing more pins to be available for control purposes. Thus the full width of the address bus appears only at cycle T2.



T1 – Lower 8 bits of address appears on data bus.
T2 – Read signal goes low, data from memory appears on data bus.
T3 – Read signal goes high, data goes into CPU.



T1 – Lower 8 bits of address appears on data bus.
T2 – Write signal goes low, data from CPU appears on data bus.
T3 – Write signal goes high, data goes into memory.

Fig 2.2 Timing diagram of a typical 8080/8085 instruction

2.5.2 Memory and I/O mapped addressing

The 8080 bus makes a distinction between addressing memory and I/O devices by having an IO/M signal. This is because the 8080 and 8085 have different instructions for accessing them as they allow for greater flexibility in accessing I/O. Firstly, memory mapped I/O allows the use of the interfacing device as if it were a memory device using standard load/store/move instructions in assembler, or high level language assignment statements. Displays are a typical example of this sort of device.

Second, I/O mapped instructions take the form of “input” or “output” instructions. A comparison between the two modes of addressing is shown in Table 2.2.

Table 2.2 Comparison between Memory mapped and I/O mapped devices

Memory mapped	I/O mapped
Uses microprocessor main memory address space	Uses separate memory space apart from processor's main memory.
May interfere with allocating memory to programs as we have to note that there are "special" locations that are not standard memory.	No concerns about special address ranges
Use flexible addressing modes to access data.	Normally all data must pass through accumulator - can be slow.

2.6 Memory Devices on the 8080 Mode Bus

2.6.1 Introduction to Types of Memory Devices

There are many different types of semiconductor memories, but we will focus on two: namely, Random Access Memory, or RAM, and Read Only Memory, or ROM. Each has its own set of technologies and uses.

RAM - Random Access Memory

The term RAM has come to mean a memory device where data can be stored as well as retrieved, and where the process of storing data at a particular location takes about the same time as the process of retrieving it from that location.

Most RAMs are volatile devices i.e. the stored data is lost when the power source is removed. However, there are non-volatile RAMs on the market which contain either a capacitor, or a small re-chargeable battery cell, so that the stored data can be retained for some time after the power is removed.

2.6.1.1 ROM - Read Only Memory

The logic function is stored in the circuit permanently without the need of electrical power to sustain the memory it is non-volatile memory). Also, the input combination has no effect on how long it takes to read the output combination random access

memory). So, a ROM can be defined as a fixed memory whose contents cannot be altered during normal operation.

Below we summarize some of the main features of ROMs, comparing them with RAMs.

	RAM	ROM
Volatile	yes	no
Random Access	yes	yes
Read	yes	yes
Write by processor	yes	No

There are a number of types of ROM, based on the methods of storing the data in the ROM, and whether or not that data can be erased.

In ROM devices, the data is stored as part of the manufacturing process, and can never be changed after that. Thus it requires costly manufacturing processes and is not economical unless many thousands are required.

Applications :

ROMs are used to store unchanging data like user prompts, error messages, character generators for video displays, or as the bootup program for computers. But ROMs are widely used in simpler forms, for example, even as a simple 4 line to 16 line decoder or a BCD to 7 segment decoder.

EPROM - Erasable PROM/One Time Programmable ROMs

An EPROM or Erasable PROM is a device in which the data can be erased after it has been stored and so it can be reused many times. The erasing is done by shining ultraviolet light on to the chip through a quartz window for 20 to 30 minutes. As a result, the entire data contents are erased each time. Because of the quartz window, the chip has to be made of ceramic. In order to make using of existing processes and lowering the cost of the package, manufacturers offer EPROMs in plastic packaging that cannot be erased. These are known as One Time Programmable ROMs (OTPROM).

Applications :

For design testing and development, it may be necessary to revise and test a program many times before it is considered error free. Because a PROM can only be programmed once, it is much too expensive for this purpose. An EPROM or Erasable PROM, which can be re-used many hundreds of times is used in this case. Although the initial cost of the IC package is greater than for a PROM, it becomes effectively much cheaper the more it is re-used. Because they are now relatively cheap. EPROMs have replaced PROMs in many applications.

PROM - Programmable ROM

To overcome the problem of expense when only small quantities are required, programmable ROMs are produced. Here, the data is not stored during manufacture, but later, in the field, by a user with the necessary programming equipment. Once stored,

the data can never be changed. The process involves the "blowing" of small nichrome fuses or links in the device, and is thus non-reversible. Hence these devices are often called fusible link PROMs. They are normally bipolar devices, requiring a different manufacturing process.

Applications :

The main use of PROMs is for small-scale production where the economies of scale are not available. Because a chip manufacturer can produce the basic (unprogrammed) device in large quantities, they are comparatively inexpensive, but the user of the PROM must then pay the additional costs of programming the device. A custom-made address decoder is a very useful application of a small Field-programmable ROM.

EEPROM - Electrically Erasable PROM

An EEPROM or E^2PROM (Electrically Erasable PROM) is a special type of erasable semiconductor memory where the data can be erased and re-written electrically in circuit. This class of devices has recently become very important as storage devices. But it is not like a RAM. The process of storing data takes very much longer than reading it, usually about 30 milliseconds writing time. A later type of EEPROM is Flash memory. The main difference is that FLASH updates its contents in terms of group of bytes or sectors whereas EEPROM does so a byte at a time. FLASH can therefore store large amounts of data relatively quickly. Also, EEPROM is available with a serial interface, which makes it smaller, both physically and in storage capacity. Parallel devices are larger on both areas, thus addressing different applications.

Flash memory can be purchased as bare memory chips, which generally use NOR gate technology used to store programs as the memory can be accessed randomly. Interfacing to the chip involves hardware design and it may not be possible to remove the bare chip, but it can take very little space. In terms of software, little effort is needed.

However using NAND gate technology, flash memory is commonly available as "memory cards". Examples are Compact Flash (CF), Secure Digital (SD), MultiMedia Card (MMC) formats and others. This is because NAND flash memory favours sequential access of data within blocks of memory such as in these devices. Hardware on the cards make interfacing easier. For example, CF cards appears like hard disks while SD/MMC cards can use the Serial Peripheral Interface (SPI), a commonly used protocol to interface to it. This can make transfer of data easier as card readers are common, but interfacing is more complicated.

Flash memory also appears in standalone portable storage devices. For example, portable music players can appear as a hard disk when attached through the USB interface.

Applications :

The form of flash memory to use is dependent on the application. As part of a system, it is natural to use flash memory for updates to programs as data stored within can be done easily and automatically. An example of this is the ability of modern equipment to update their firmware by "flashing" an update from the manufacturer.

However, if we wish to use flash memory for storing of large amounts of data, it makes more sense to use a CF card and access it like a hard disk.

Serial EEPROM is mainly used to store settings and configuration information where its low cost, low power consumption and small size are advantages here.

This illustrates the fundamental difference between RAM and EEPROM. In a RAM the processes of storing and retrieving data take about the same time - in the order of tens to hundreds of nano-seconds. While an EEPROM can be read from in about the same time, the process of storing data takes much longer.

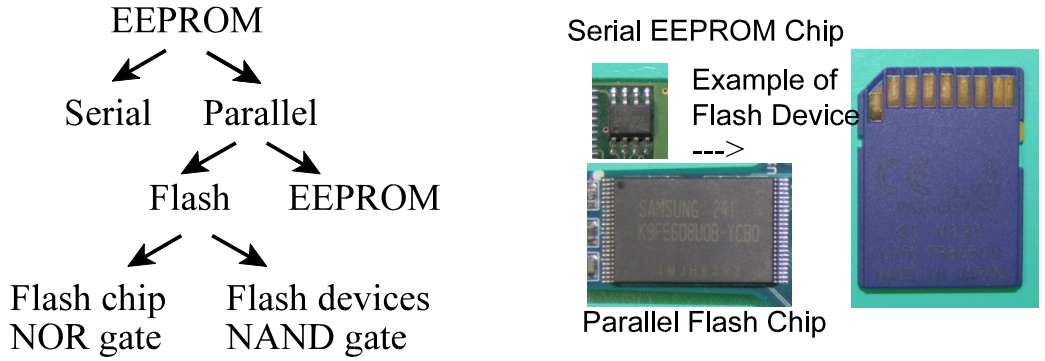


Fig 2.3 EEPROM family

In summary, the diagram describes the EEPROM family. The type used will depend on the applications.

2.6.1.2 RAM Devices

RAM devices can be classified into two types: static and dynamic. RAM's are manufactured using both MOS and bipolar technologies. MOS uses NMOS and CMOS. These are usually simpler devices, and yield higher densities than bipolar process. Bipolar devices include Schottky TTL, ECL, and IIL types. Usually faster than MOS devices, and accordingly have higher power dissipation.

Static RAM's

Static RAM's use a flip-flop as the storage element, one F/F for each bit of data storage. So an 8K RAM has 8192 F/Fs plus the necessary address decoding and I/O buffering circuits. Information is retained in the F/F for as long as power is applied.

Dynamic RAM's

Dynamic RAMs use the ability of MOS capacitors to store charge as the basic storage element. 1's or 0's are indicated by the presence or absence of charge on the capacitor. If the capacitor is charged above a certain level (that is, its voltage is above a certain threshold), it represents a logic 1, otherwise it is a logic 0. An ideal capacitor, once charged, maintains a constant voltage, but charge will leak away in a practical capacitor, and so the voltage will change.

The capacitor's voltage (the data) must be read regularly, and if it is a logic 1, it must be restored to full voltage, before too much charge leaks away. This is called refreshing. If a dynamic RAM is not refreshed often enough, the voltage of the logic 1's will fall below the threshold and all stored bits of data will revert to 0's. For example, an 8K dynamic RAM contains 8192 MOS capacitors plus associated decoders, buffers, etc. All these capacitors must be refreshed regularly, typically every 2 milliseconds.

2.7 I/O Devices on ISA bus

A more common use of plug-in cards is to provide many more ports to be used for interfacing with external devices. These ports normally interface with devices that are much slower than the main processor so that the relatively slower speed of data transfer is not an issue. As we saw earlier, the Intel architecture uses 16 bit addressing for I/O. On the PC, only 10 address pins are actually used, to reduce the hardware decoding requirements.

2.7.1 BASIC INPUT/OUTPUT HARDWARE

Various hardware elements can perform some of the functions of the I/O section. Simple devices include flip-flops (latches) and buffers. In this chapter we will look at some of the basic elements used in interfacing circuitry.

2.7.1.1 Buffers

Most microprocessors are MOS products, hence the driving capabilities of such devices

are usually restricted to at most 1 TTL load or 5 MOS loads. This severely restricts the operation of the microprocessor in driving other devices, for example, RAMs, ROMs and other support devices. In order to increase the driving power, or driving current, buffers are used.

Buffers are TTL ICs which allow a signal to pass through them without altering the logic value of the signal. However, the buffer increases the DC drive characteristics of the signal. Hence, the microprocessor, with the aid of a buffer is capable of driving many more circuits.

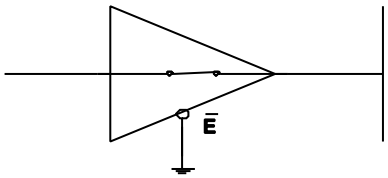
Another advantage of using buffers is that the microprocessor is isolated from any outside signals. If there is a change in voltage along the signal lines, the buffers are first affected, thereby preventing any damage to the microprocessor. Being mainly TTL ICs, they introduce a small delay in the transmission of data.

Two variations of the buffer exist. The first is simply a one-in one-out buffer which takes an input and increases the driving current of the signal at the output for all logic levels.

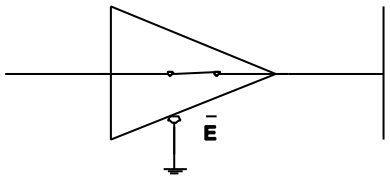
The second type of buffer is called a Tri-state buffer. This device has an additional control signal. When the control is active, the device performs exactly as a

buffer. When the control is de-activated, the device prevents any output from appearing. The output of the device turns into a high-impedance state giving neither a logic 1 or 0. The table below shows the logic states of such a buffer :

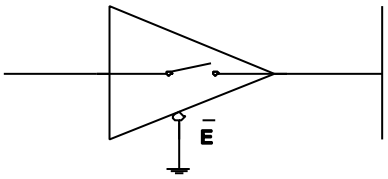
Input	Control	Output	State
0	Active	0	buffers input
1	Active	1	buffers input
0	Non-active	High impedance	Tri-state
1	Non-active	High impedance	Tri-state



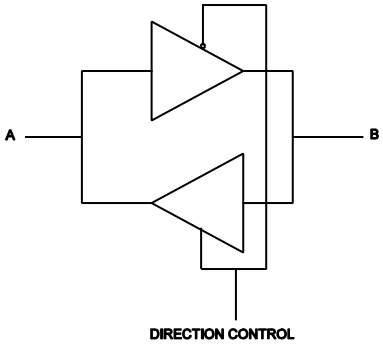
Tri-state buffers are useful in circuits which only allow the passage of signals when a particular state is active. This state can be connected to the Control input of the buffer.



A variation to the tri-state buffer is the bi-directional buffer. This buffer allows the passage of data from one point to the other depending on a Direction Control Signal. This allows a system to control the direction of flow of data from one point to the other. The figure below shows the operation of such a buffer.



When the Direction Signal is ‘low’, data flows from Point A to Point B. When the signal is ‘high’, the data flows from Point B to Point A.



A typical application of the bi-directional buffer is in the buffering of the microprocessor's data bus. Using the _RD line as the Control Signal, the microprocessor is able to control the flow of data into the microprocessor only when data is required. At all other times, the direction of flow of data is from the microprocessor to the other devices.

2.7.1.2 Latches

Latches are mainly made up of D-type flip-flops. The latch takes in a signal via a strobe (clock pulse). The latch then holds the signal until another strobed signal is applied or until the power is removed.

The latch has an advantage over the buffer in the sense that the signal is held at the particular logic level until a change is requested. The buffer holds the logic level as long as the signal is also at that level. Latches are thus used in preference to buffers in situations where the receiving end requires a little more time to process the signal. The figure below shows the operation of different latches in response to changing signals.

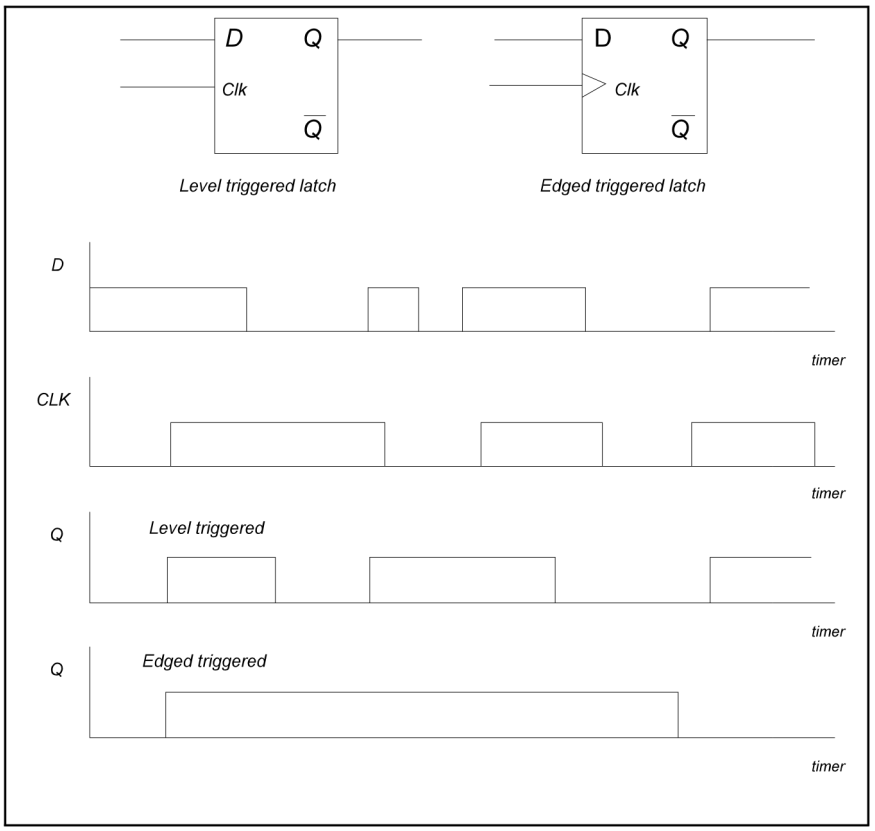


Fig 2.4 Signal Response of Latches

Latches can be level-triggered (changes output as long as the strobe is active) or edge-triggered (changes output only on either a rising or falling edge of the strobe signal). In cases where ambiguity is to be avoided, edge-triggered latches are used.

2.7.1.3 Input/Output Ports

Basic Input Ports

The basic input port is a tri-state buffer connected to a data bus line. The control signal of the buffer is usually the combination of _IORD and the selected port address, which is obtained using address decoding.

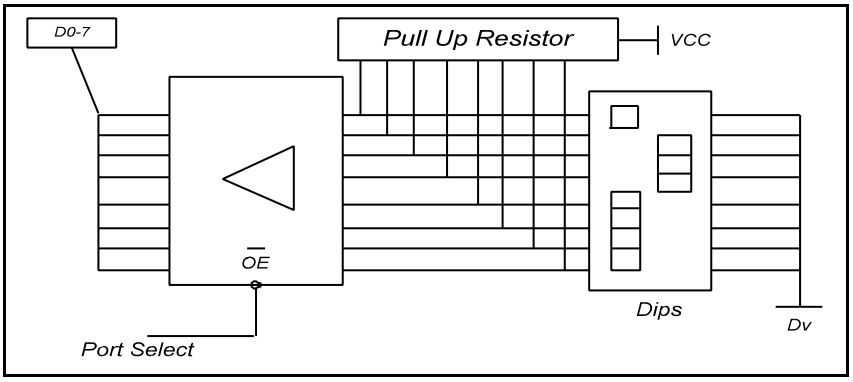


Fig. 2.5 An Example of an Input Port using a Buffer

Basic Output Ports

The basic output port is a latch. The CPU places the necessary data to be output on the data bus lines and then clocks the data into the latch with the control signal which is usually the combination of the _IOW and the selected port address, which is obtained using address decoding.

The latch is preferred over the buffer as external devices are usually slower than the CPU. Hence, the port has to hold the data until the device is ready to read it.

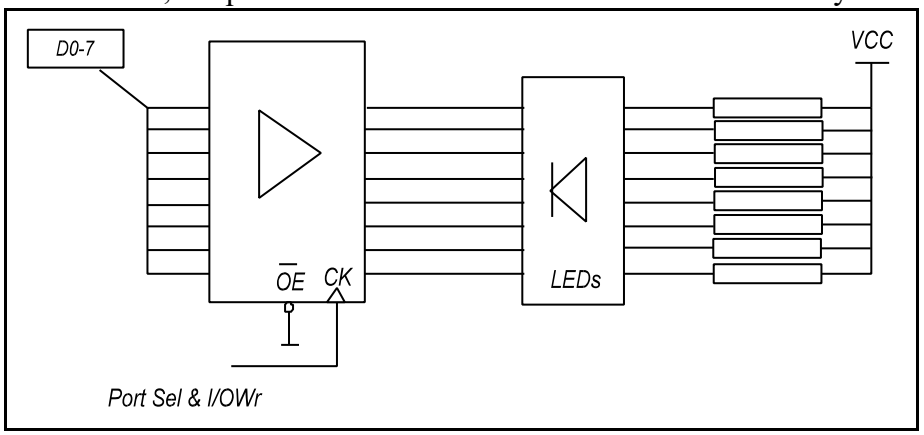


Fig. 2.6 An Example of an Output Port using a Latch

2.8 The BIOS and boot up process

As a final note and application, we see how ROM is used in a typical microprocessor where memory is mainly DRAM. We will also see how a basic "monitor" program considerably enhances the use of a bare computer system.

The 8080 follows a given sequence when it is RESET. Recall that ROM is non-volatile, so that if a processor starts, it is guaranteed to have valid code when it jumps to that location.

In most other microprocessors, what happens after that is purely a matter of what the programmer puts into the ROM code. Starting from the early microprocessor systems available, the vendor would often put in basic routines in a so called "Monitor Program" and include it in the ROM.

These allowed users to do fundamental tasks like simple output to the screen, keyboard handling, output to printer, serial communications and so on. This allowed users to quickly bring up their system to productive use and allowed a measure of standardization to programs as they used common routines to do input and output.

In some systems – typically PCs, this monitor program is known as the Basic Input Output System or BIOS. More routines could be added, for secondary storage devices like hard disk.

A BIOS may occupy as much as 64K of ROM which is not practical for small processors, but monitor programs can be as small as 2K. However if the system is simple enough, a programmer can make do without any built in monitor program. Consider the boot up process for the 8080 processor:

Microprocessor function

i) Performs a jump to location 0H.
ii) Executes the code found there, which is normally contained in ROM.

Monitor program (BIOS) function:

i) It searches for a secondary storage device like a floppy or hard disk.
ii) It reads in the boot sector (normally 512 bytes in length) for the device.
iii) Puts the contents into RAM.
iv) Jumps to that code and executes it.
v) This code will have instructions to load other larger files which will continue to initialise the system.

This process is called "bootstrapping". Other processors will jump to other locations upon start up but the startup process follows the same sequence.

3 Decoding on a bus system

Although the 8080 bus has 16 bits for addressing, we first consider the use of these 16 bits for memory chips, then the first 6 bits for I/O devices. To see what is needed, we will examine the interfacing of some memory devices.

3.1 Accessing a memory device

As we have considered the address, data and control busses from the processor point of view, now we consider them from the *memory* point of view.

3.1.1 Memory address, data, control bus

In contrast to the processor address bus, the memory address bus will tell a memory device which particular location is being accessed. The number of memory address lines required depends on the size of the memory device. (e.g. a device with 2K (i.e. 2048) locations, requires 11 address lines, because 211 = 2048).

In the same way, the quantity of data stored at each memory location depends on the particular memory device. It may be a single bit, or 4, 8 or 16 bits per location. Units of memory may be connected in parallel to meet the processor data width. For example, four 8 bit memory devices may be connected in parallel to a 32 bit processor.

3.1.2 Steps to access memory

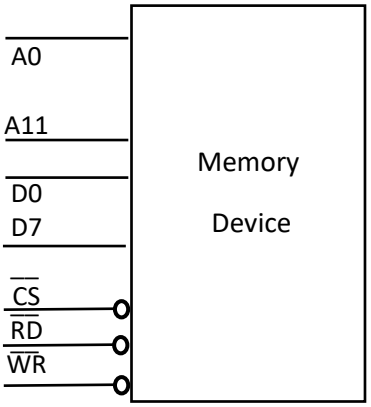


Fig 3.1 Steps to access memory

The block diagram of a memory device is as shown. To access data, the following steps are executed.

1) The address is placed on the address bus. The lower bits will be used by the memory device for internal decoding, the higher bits go to a decoder chip which will generate a chip select signal ($\overline{CS}$). This will then select one of the several memory devices.

2) READ

If doing a read operation, the processor will generate a read signal ($\overline{RD}$) which is connected to the output enable ($\overline{OE}$) of the memory chip. The memory device will place the data from the memory location after a short delay. The processor then reads in the data.

3) WRITE

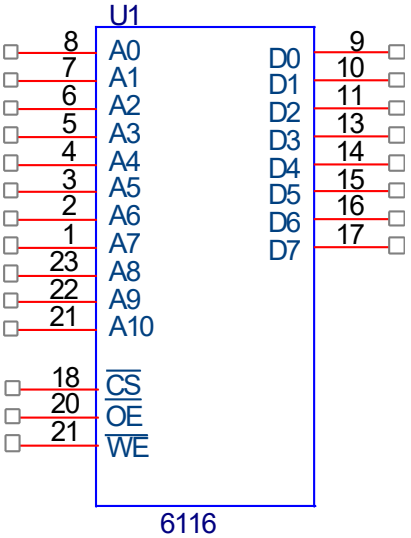
If doing a write operation, the processor will places the data to be written on the data bus, then generates a write ($\overline{WR}$) signal, which is connected to the write enable ($\overline{WE}$) of the memory chip. The memory device will then latch the data in.

3.1.3 Partial and Full Address Decoding

In most practical situations, the devices connected to a processor address bus require fewer address bits than there are bits provided on the address bus. The 8080 has 16 pins while a 6116 RAM has only 11 pins for addressing. Since there are 5 pins unused by the RAM chip, then this is a case of *partial address decoding*. If we have a memory system where all the address pins are used in memory access, we then have *full address decoding*.

As an illustration of how partial and full address decoding are implemented, we shall look at how a memory chip can be connected to a 16-bit address bus.

3.1.3.1 A memory device



Pin Name	Function
A0 to A10	Address input
D0 to D7	Data input/output
CS	Chip select
WE	Write enable
OE	Output enable
Vcc	Power supply
Vss	Ground

Fig 3.2 6116 schematic

Pins A0 to A10 are the 11-bit address inputs to the 2048 locations in the RAM while D0 to D8 are the 8 data bits in each location. CS1 is the chip select inputs (active high or low depending on whether there is a bar on top) that enable the chip.

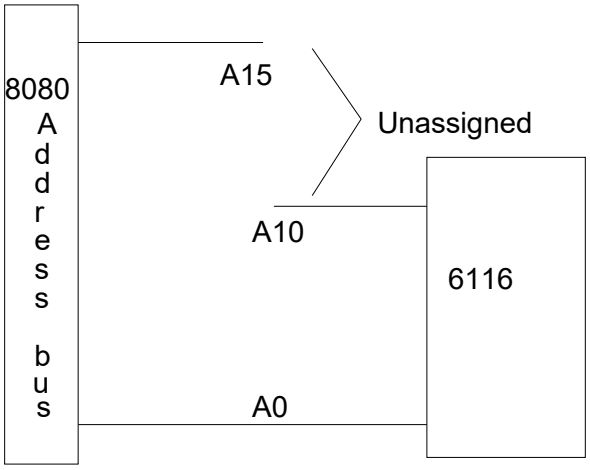


Fig 3.3 Foldover

When the 6116 is connected to the 8080 address bus as shown only the states of the address lines A0 to A10 will determine which location inside the RAM is accessed. The other address lines A11 to A15 are ignored by the chip. Hence the location 000 0000 0000B can be accessed by the addresses XXXX X000 0000 0000B. Therefore addresses from the processor bus 0000H, 0800H, 1000H and so on will all access the same memory location 0000H.

This is an inefficient way of using the address bus since even though 216 = 65536 locations can be selected individually, only 2K are used, each responding to 65536/2048 = 32 different processor addresses.

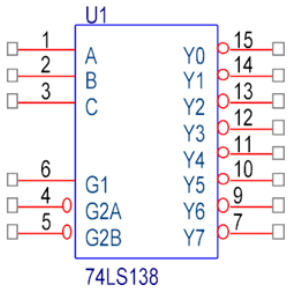
Memory Foldover

When a memory address (or group of physical addresses) responds to several different *processor* addresses, the effect is known as memory foldover. In the example above, there is a 32 times foldover.

In small systems, this is tolerable. But we see two problems here. First is that if more than one device needs to be attached to the bus, how do we fit it in? Secondly is that a programmer may access data from the wrong memory location.

When connecting several memory ICs to an address bus, we can use logic gates as address decoders using K-Maps or standard Boolean logic. However, it is much easier to use demultiplexers or decoders, because of the way memory is laid out, which in many cases is in sequential blocks of addresses, as we will see shortly.

One of the most commonly used demultiplexers is the 74138 1-of-8 demultiplexer. Looking at the output pins, note that the pin that goes low will correspond to the binary input at pins A, B and C. For example, an input of 111 will cause pin Y7 to go low.



PIN No	SYMBOL	NAME AND FUNCTION
1,2,3	A,B,C	Address Inputs
4,5	_G2A,_G2B	Enable Inputs
6	G1	Enable Inputs
7, 9 to 15	Y0 to Y7	Outputs
8	GND	Ground (0V)
16	Vcc	Positive Supply Voltage

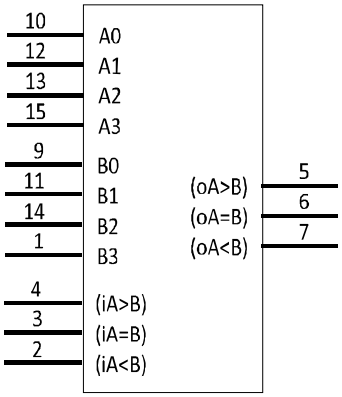
Fig 3.4 74LS138 schematic

Truth table

INPUTS						OUTPUTS							
ENABLE			SELECT										
G2B	G2A	G1	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	X	L	X	X	X	H	H	H	H	H	H	H	H
X	H	X	X	X	X	H	H	H	H	H	H	H	H
H	X	X	X	X	X	H	H	H	H	H	H	H	H
L	L	H	L	L	L	L	H	H	H	H	H	H	H
L	L	H	L	L	H	H	L	H	H	H	H	H	H
L	L	H	L	H	L	H	H	L	H	H	H	H	H
L	L	H	L	H	H	H	H	H	L	H	H	H	H
L	L	H	H	L	L	H	H	H	H	L	H	H	H
L	L	H	H	L	H	H	H	H	H	H	L	H	H
L	L	H	H	H	L	H	H	H	H	H	H	L	H
L	L	H	H	H	H	H	H	H	H	H	H	H	L

We see that the 74138 can select one of eight possible devices based on its ABC inputs. If we add this to the thirteen inputs of the 6116, we see that we now have fourteen (11+3) inputs, which still causes foldover.

To take care of this, we can still choose to use logic gates, but we introduce another device, the 74LS85 4 bit comparator which makes decoding more versatile.



PIN No	SYMBOL	NAME AND FUNCTION
3	iA=B	Input A=B must be 1
10, 12,13,15	A	A word input
9,11,14,1	B	B word inputs
6	oA=B	equal to output
8	GND	Ground (0V)
16	Vcc	Positive Supply Voltage

Fig 3.5 74LS85 schematic

TRUTH TABLE

COMPARING INPUTS				CASCADING INPUTS			OUTPUTS		
A_3,B_3	A_2,B_2	A_1,B_1	A_0,B_0	$I_{A>B}$	$I_{A<B}$	$I_{A=B}$	$O_{A>B}$	$O_{A<B}$	$O_{A=B}$
$A_3>B_3$	X	X	X	X	X	X	H	L	L
$A_3<B_3$	X	X	X	X	X	X	L	H	L
$A_3=B_3$	$A_2>B_2$	X	X	X	X	X	H	L	L
$A_3=B_3$	$A_2<B_2$	X	X	X	X	X	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1>B_1$	X	X	X	X	H	L	L
$A_3=B_3$	$A_2=B_2$	$A_1<B_1$	X	X	X	X	L	H	L
$A_3=B_3$	$A_2=B_2$	A_1=B1	$A_0>B_0$	X	X	X	H	L	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0<B_0$	X	X	X	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	H	L	L	H	L	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	L	H	L	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	X	X	H	L	L	H
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	H	H	L	L	L	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	L	L	L	H	H	L

We will use only the A=B output. From the truth table, when the digital value at the A inputs equal the digital value at the B inputs, and IA=B is high, then the output OA=B pin will go low.

What makes this device versatile is that we do not need inverted versions of address signals, which is true if we use the 74138. We can set the desired bit pattern at either the A or B inputs.

Another important point is that there are other devices connected to the system that use up certain parts of the memory map. Thus we need flexibility in locating the starting address of our memory block.

EXAMPLE: Design a 16K memory space starting from 8000H, using 6116 memory chips.

In designing a full decoding system using a 74138, we should note the number of address lines are used by the memory chip itself. This is because these lines cannot be used for decoding. Thus only the remaining lines on the host processor address bus is available.

To summarize, we may use the following steps:

1) Identify how many address pins the memory chip uses.
2) Calculate the number of chips that are required in the design.
3) Draw the memory map of the design. Do this by laying out the chips contiguously, that is, the starting address of a chip follows immediately after the last address of the previous chip. Write down the range of addresses of each chip in hexadecimal.
4) Draw a truth table. This is the binary representation of the range of addresses.
5) Looking at the bits, observe any pattern of numbers, for example, running binary numbers in the bits.
6) Fit this pattern to the truth table of the 74138. First, see which address pins are to be assigned to inputs A, B and C. Then see how to assign the remaining address pins to the Enable inputs of the 74138.
7) If there are unassigned *processor* address pins, assign them to the 74688. 8) Finally, produce the schematic.

Applying to the above example:

1) The memory device to be used is a 6116, which is a 2048 by 8 bit device. This device uses address pins A0 to A10. (2^{11} = 2048). Each chip will take up 2048_{10} or 7FFH addresses.
2) Since we require 16K bytes in the design, we need 8 chips. (16K/2K).
3) This 16K will occupy the range from 8000H to BFFFH. So the *first* memory chip will start from 8000H and occupy the address range 80000 - 87FFH. We continue until we get the following memory map:

RAM 1	-	8000H to 87FFH
RAM 2	-	8800H to 8FFFH
RAM 3	-	9000H to 97FFH
RAM 4	-	9800H to 9FFFH
RAM 5	-	A000H to A7FFH
RAM 6	-	A800H to AFFFH
RAM 7	-	B000H to B7FFH
RAM 8	-	B800H to BFFFH

To draw the truth table, let us look at the first chip. It occupies the address range 8000-87FFH. Using binary, and noting that the MSB is address line A15:

```
                    MSB
```

The range in binary is: `1000 0000 0000 0000 (8000H)`
`1000 0111 1111 1111 (87FFH)`

We are not interested in the address lines used by the chip at this point. Note that the 0's and 1's change only at lines A0 to A10.

In order to simplify the design process, we will introduce a notation. Instead of writing two lines with all the 0's and 1's, we will represent the above range by:

```
A15         A10        A0
 1 0 0 0 0 X--------X 8000 - 87FFH
```

"X" means that the value for the address line can be 0 or 1. The dashed line "X---X" means that address lines from A0 to A10 can be 0 or 1. So they are not available for general use. This highlights the fact that only address lines A15 to A11 are available for decoding. Proceeding in similar fashion for the other chips, we have the following truth table.

```
A15 A14 A13 A12 A11 A10 -- - - - - - - - - A0
1   0   0   0   0   x------------------------x  8000-87FFH
1   0   0   0   1   x------------------------x  8800-8FFFH
1   0   0   1   0   x------------------------x  9000-97FFH
1   0   0   1   1   x------------------------x  9800-9FFFH
1   0   1   0   0   x------------------------x  A000-A7FFH
1   0   1   0   1   x------------------------x  A800-AFFFH
1   0   1   1   0   x------------------------x  B000-B7FFH
1   0   1   1   1   x------------------------x  B800-BFFFH
```

1. Looking at the truth table, we can see the address lines A13 to A11 taking on values that increment, starting from 000 to 111. If we assign these lines to a 74138, we can enable up to 8 devices one at a time, depending on the values of A13 to A11.

2. Comparing this with the truth table of the 74138, we see that A13 should be assigned to input C, A12 to B and A11 to A. Note that this incrementing bit pattern is only true for this situation. Other situations will have different patterns at different address lines. The important thing is to look out for a pattern to fit to the truth table of the 74138.

3. It is convenient to let A15 to A14 be enabled by the 74LS85, so it activates when it has the value 2H.

The final design is shown below:

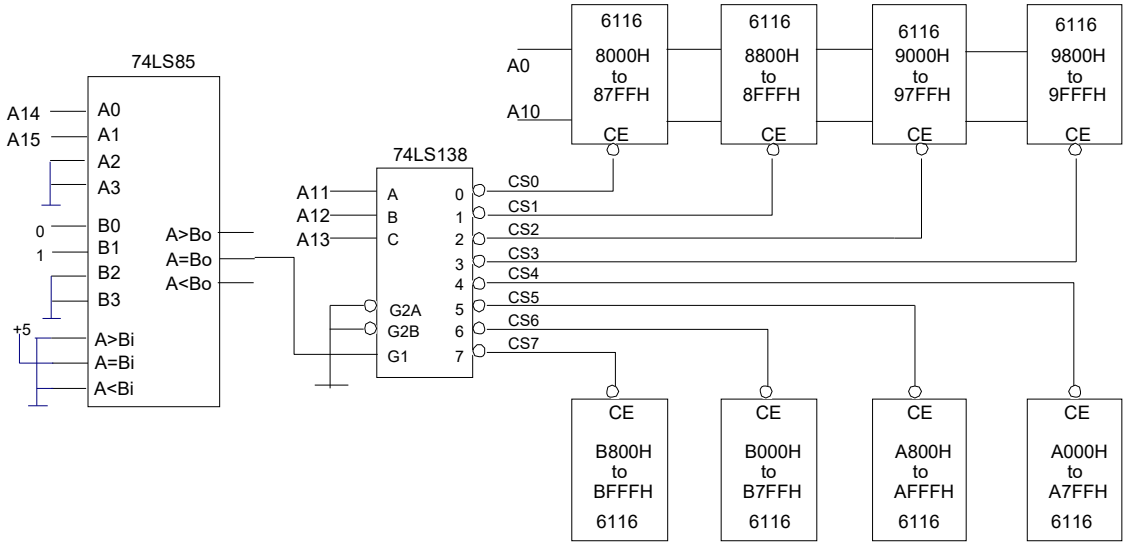


Fig 3.6 Circuit which memory maps 16KB of SRAM starting from 8000H

3.2 Input / Output Decoding

The 8080 is able to access buffers and latches, but now we consider the use of SMI bus to do so. As we have seen, memory devices use only some of the available address lines. The lower address lines will select a single memory location in the memory device. The higher address lines from the system are used by a decoder to select one of several memory devices. Also, the memory device will have an Output Enable pin for a read and a RAM device will also have a Write Enable pin.

3.2.1 Single address decoding

However, buffers and latches have only a single address location. The PC architecture has used up quite a few I/O addresses so that there are few left for the user. Also, other devices connected to the system may have used up other address.

Thus we need to decode each device exactly, so we do not activate other I/O by accident. What this means is that for a 16 bit I/O address space such as that available on Intel microprocessors we need to use all address lines, giving us theoretically 64K addressable I/O devices. But we note that the SMI implementation only uses 6 bits, making the hardware requirements somewhat simpler.

Also, since buffers have only one control pin (normally the Enable pin) and latches use the clock pin, we need an external gate to control the access, so that processor reads are directed to buffers and writes are directed to latches. Of course, if there is no confusion as to the assignment of addresses, the gates may not be required.

Example: Design an input port of address 30H and an output port of address 31H.

Truth Table

```
A5 A4 A3 A2 A1 A0
1  1  0  0  0  1    31H
1  1  0  0  0  0    30H
```

We let A0-A2 be decoded by the 74138. The bit pattern on the 'B' inputs correspond to bits A3 to A5.

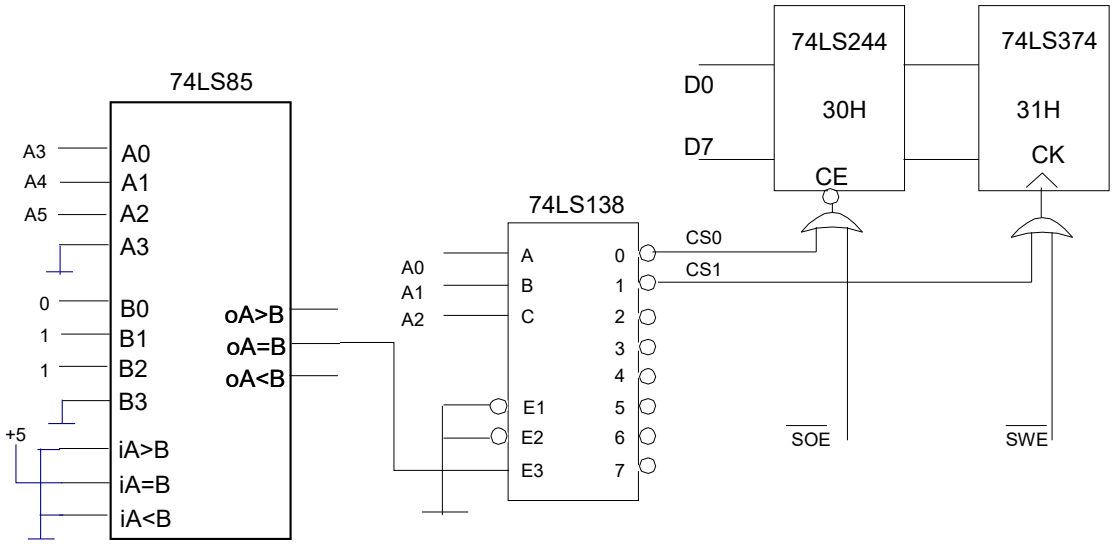


Fig 3.7 I/O decoding with 74LS85

As we see, the 74LS85 in conjunction with a 74138 allows us to decode individual I/O addresses. The 74LS85 activates a block of 8 devices through the 74138. The *starting* address of this block will have the A0-A2 pins at logic 0. This address is known as the *base* address as all the other I/O addresses are simply added to the base to get its address.

3.3 Summary

In summary, we saw how memory and I/O devices may be decoded on the 8080 and SMI busses respectively. The TTL chips 74LS85 and 74138 helped us to easily and flexibly do this. The main differences between memory and I/O is that:

i) Memory devices contain many storage locations within itself. Selecting a memory chip through a decoder only enables the chip. Internal decoding circuitry *within* the memory chip will select individual memory locations.

Buffers and latches, being individual devices, need full decoding to be done externally as it does not have the circuitry internally.

ii) In addition to enable pins, memory devices have control pins which determine whether data is written or read to.

Buffers and latches are normally activated through a single control pin. Again, external gates may be needed to correctly decode the signal.

Some examples of current devices that use the 8080 bus are given in the following online links.

https://www.st.com/resource/en/application_note/cd00200423-using-the-highdensity-stm32f10xxx-fsmc-peripheral-to-drive-external-memories-stmicroelectronics.pdf

https://www.nxp.com/docs/en/supporting-information/FTF-ACC-F1179_Introduction_to_FlexIO.pdf

https://www.raspberrypi.org/documentation/hardware/computemodule/datasheets/rpi_DATA_CM3plus_1p0.pdf

4 Keyboard Interfacing

4.1 Introduction

A very common method of input to microprocessors from humans is a keyboard. At a low cost, a large amount of information may be input. A key works like a single pole, single throw switch. With such switches, it is better to detect the presence of a low voltage level, rather than a high. This is due to noise. Smaller keyboards up to 16 keys or of a size that fits into a palm, are known as keypads. A keyboard consists of pressure- or touch-activated switches arranged in a matrix. To detect which key has been pressed, we use a combination of hardware and/or software means.

Two basic types of keyboards are available: encoded and non-encoded. Encoded keyboards include the hardware necessary to detect which key was pressed and to hold that data until a new key stroke. Non-encoded keyboards have no hardware and must be analyzed by a software routine or by special hardware. This is the type we will be looking at here.

4.1.1 Reading from a keyboard

The complexity of the processor task when reading from a keyboard depends on the number of keys to be detected, the input ports available on the processor and the amount of processor time dedicated to this task.

4.1.1.1 Simple keyboard reading

For a small number of keys say for 1 to 8 keys, we may assign one key to each buffer or port line. To read the keyboard, it is connected directly to a buffer. Or it may be connected directly to a port. We simply read the keys, checking for a low level.

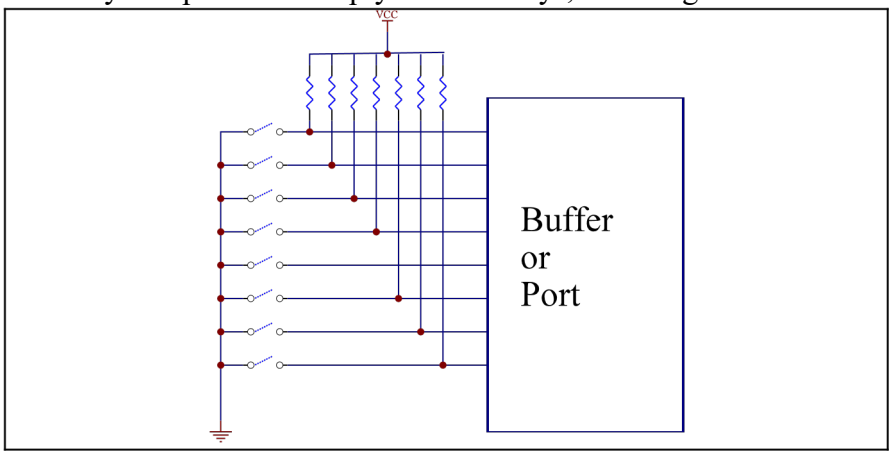


Fig 4.1 Simple key input

4.1.1.2 Matrix organisation

For larger number of keys, the keyboard may be arranged in a row and column fashion, with an n by m key organization. This is also known as a matrix organisation.

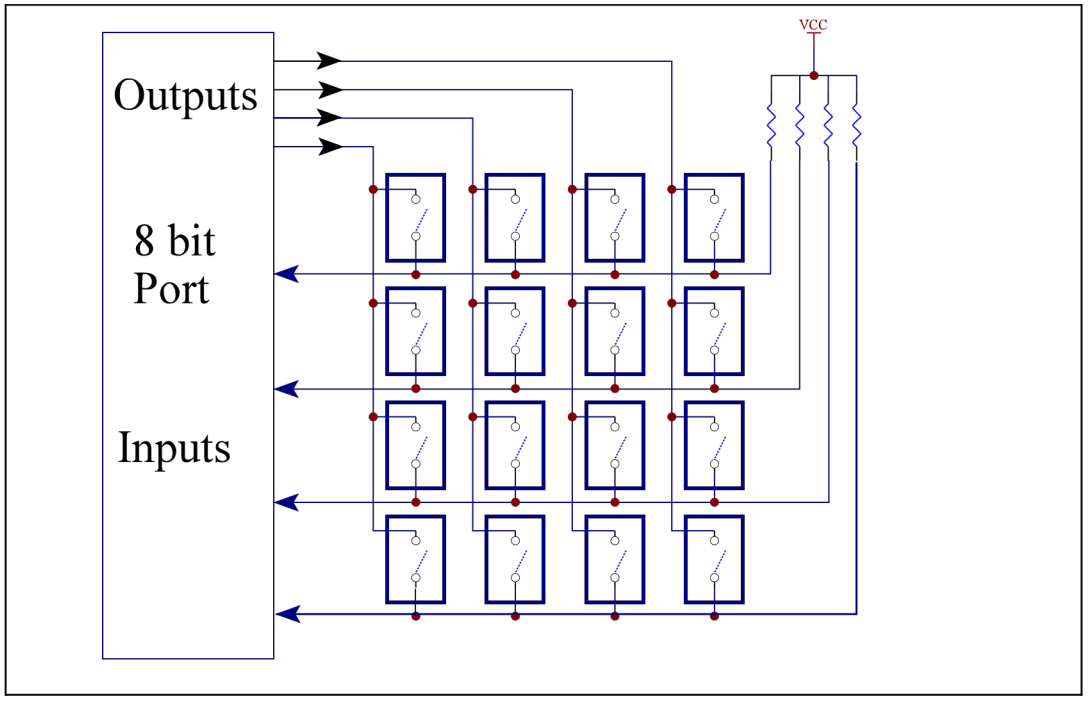


Fig 4.2 Walking Zeroes Keyboard Decode

A 16 key device can be decoded with 4 input lines and 4 output lines. This is an optimal number for an 8 bit controller as this will use one port. However, this port must be capable of having four of its bits as input, and the other four as output.
We will consider an example where the upper four bits of the port are used as output pins and the lower four as inputs. With respect to the diagram above, we denote the output lines as columns and the input lines as rows, for convenience. When the processor is not checking the keyboard, it will output all 1's. The inputs will also read all 1's, because it is tied to the pull up resistors.

Scanning the keypad

We can output to the column lines with a "walking zero" pattern and sense the row lines to see if a low voltage is detected. This key identification technique is known as "row scanning."

With respect to the diagram, we output a “0" to each of port pins 4,5,6 and 7 one at a time. We read in the input port and check for a “0" in port pins 0,1,2 and 3.

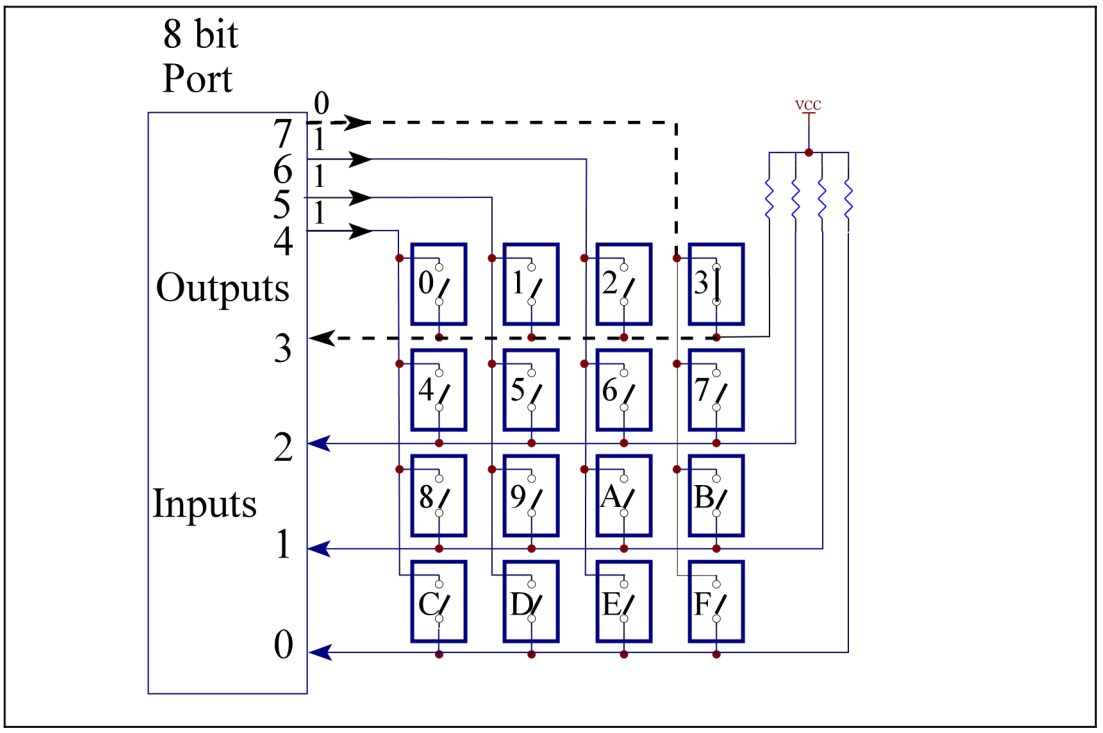


Fig 4.3 Walking “0" actual example

Let’s say the ‘3' key is pressed. If the processor did not output a “0" to pin 7, the keypress will not be registered - that is, port pins 0 to 3 will still read “1". Only when pin 7 outputs a “0" will pin 3 register a “0" as well.

Larger keyboards require more select and sense lines. First we can have latches to output the “walking zero” data to the column lines. Buffers are used to read the row lines. When selecting the key organization, we should let the smaller number be assigned to the row lines. This is because we detect a key press through these lines, the status of which is read into a register. Then software routines, most often rotates are used to detect which column has a “0". This is time consuming.

(Note: The assignment of which lines are “rows” and “columns” is entirely arbitrary. In this course we refer to the vertically oriented lines as columns purely for ease of reference)

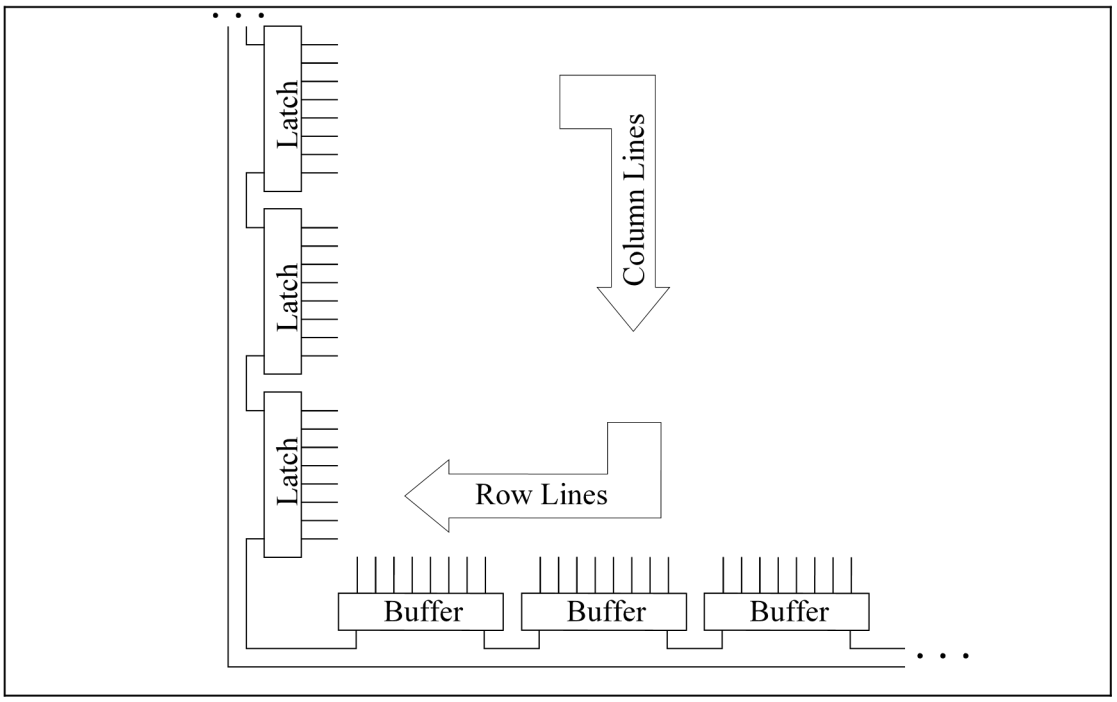


Fig 4.4 Large keyboard detection using latches

Since each column line is activated one at a time, it is possible to substitute the latches with decoders. In this case only N lines from the processor will support 2^N columns. As shown below, three 3-to-8 decoders can handle 24 column lines using 5 address lines. But up to 32 rows can be decoded.

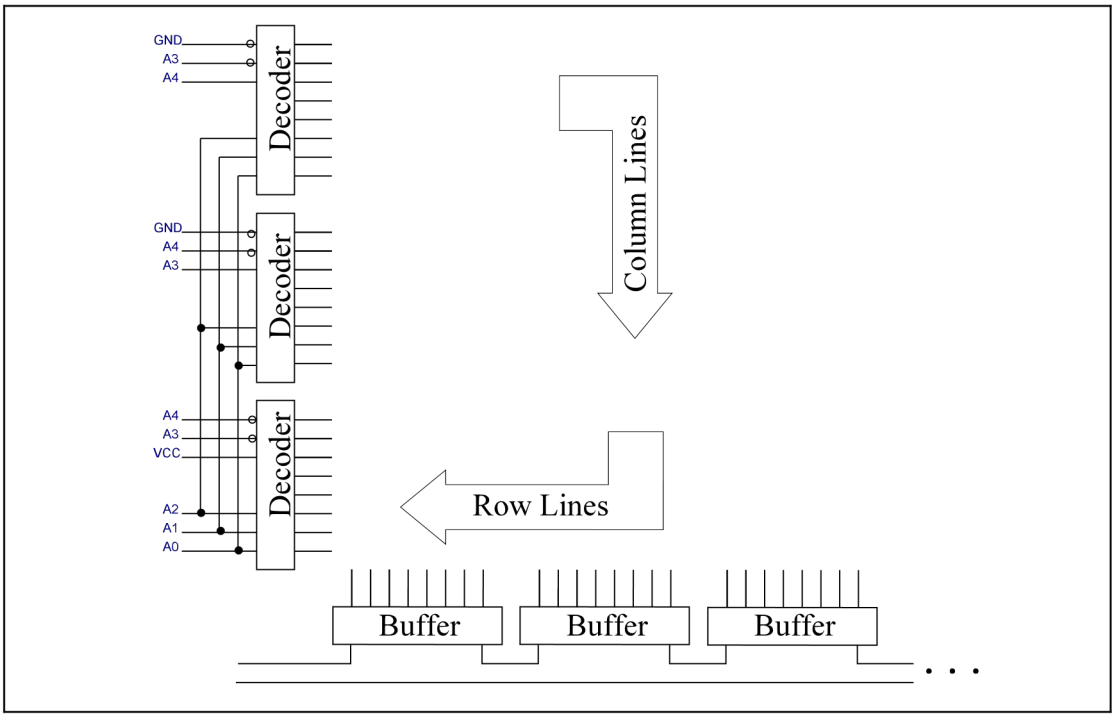


Fig 4.5 Decoder Scanning

4.1.2 Interrupt Driven Keyboard

From the previous discussion, we see that detecting a key press is a processor intensive process. We must continually select a row line and then read the column lines. If this is a matter of concern, dedicated keyboard controller hardware is available. These are normally interrupt driven and have various features built into them. But this will increase product cost.

A simple solution would be to see if we can make all the column lines go to “0" volts. If no keys are pressed, all the row lines will present a “1" level. We would only need to AND all the row lines together. A key press would then signal an interrupt, and normal scanning can proceed to find out which key is pressed.

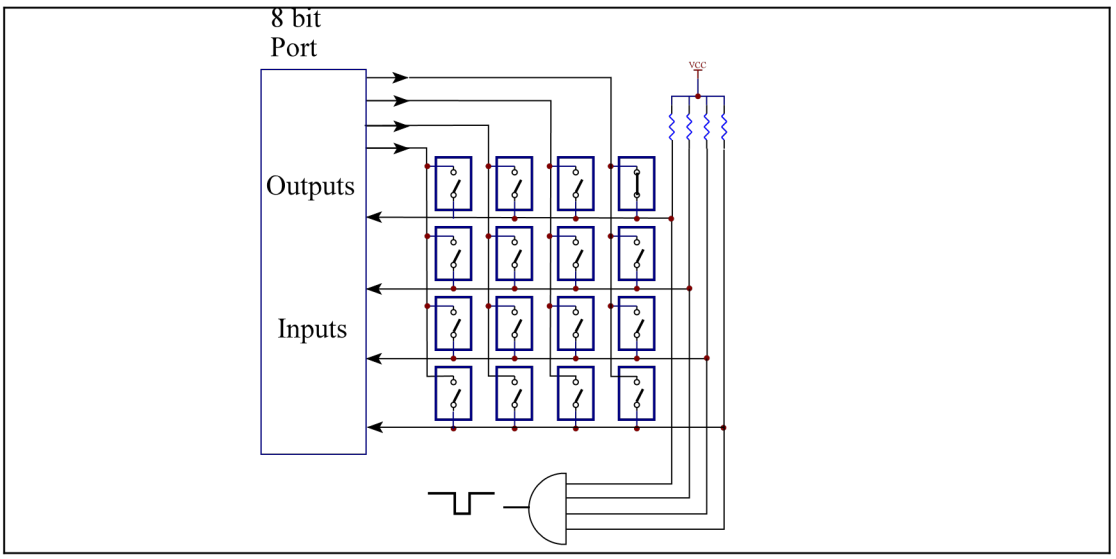


Fig 4.6 Interrupt driven keypad

4.2 Common Problems

The most common problem is that of contact bounce. In addition, there are other problems caused by certain usage patterns. In some cases, in order to increase the number of usable keys, two or more keys have to be pressed together, for example Control and Alternate keys together with another key. Or sometimes when the user presses the keys very quickly, and the processor is busy, it seems that three or more keys are depressed. These problems have to be addressed, especially when the order of digits is important. If a student’s score is 290 marks and if the scanning algorithm detects 3 simultaneous keystrokes it may read it as 092!

4.2.1 Bounce

Keybounce refers to the fact that when the contacts of a mechanical switch close, they bounce for a short time before staying together. This is also true when the switch opens. What happens is that the resistance of the switch changes, thus presenting differing voltages to circuitry for a short while. Fig. 4-7 is a time-versus-resistance plot of a typical switch contact. The bounce lasts for 10 to 20 millseconds and is denoted by the oscillatory waveform.

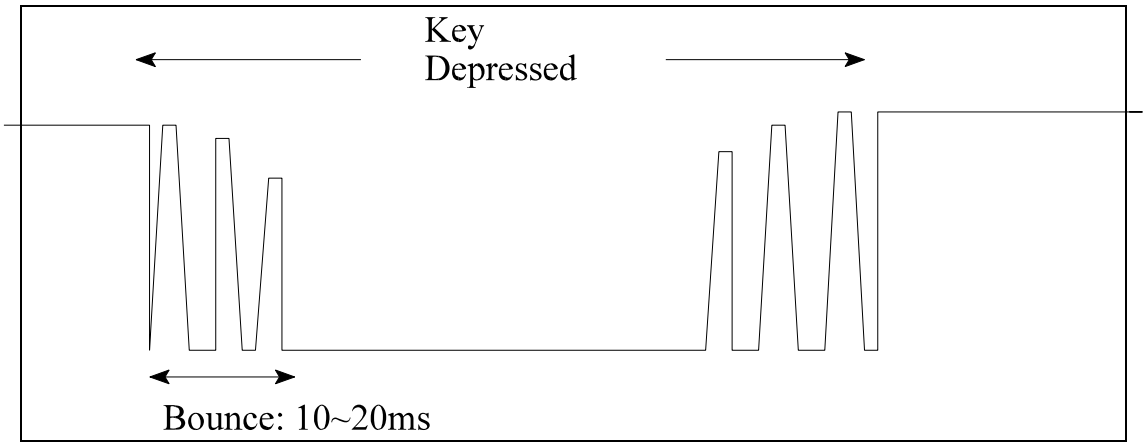


Fig 4.7 Keybounce

The solution is to wait for the status of a key to remain stable for perhaps 20 milliseconds. This may be done by hardware filtering or by a software-delay routine. The hardware solution uses an R-C filter and requires the same circuitry for each key. This is useful for if we have only a few switches in a system. In the case of a larger number of keys, software is often used.

4.2.2 Multiple key presses

When more than one key is held down at the same time, it is essential to detect this fact to prevent wrong codes from being generated. The two main techniques used to resolve this problem are the n-key rollover, and the n-key lock-out.

4.2.2.1 N-key rollover

This is again has two techniques: the software simply ignores the reading from the keyboard until only one key closure is detected. The last key to remain pressed is the correct one. This method is normally used when software routines are used to provide keyboard scanning and decoding. If the order of keys pressed is not crucial, it is possible, with more sophisticated software to store all closures in an internal buffer.

The second method is used by hardware devices. Second and later key closures are prevented from generating a strobe until the earlier ones are released. This is realized by an internal delay mechanism which is latched as long as other keys are pressed.

4.2.2.2 N-key lockout

N-key lock-out takes into account only one key pressed. Any additional keys which might have been pressed and released do not generate any codes. By convention, it may be the first key pressed which will generate the code, or else the last key pushed. The system is simplest to implement and the most often used. However, it may be objectionable to the user, as it slows down the typing: each key must be fully released before the next one is pressed down.

4.2.2.3 Phantom Key

A significant cost of n-key rollover protection is that most systems need a diode in series with every key in order to eliminate the problem created when three adjacent keys at a right angle are present. This increases the cost very significantly and is seldom used on low cost systems.

The effect is that another key seems to be pressed in addition to the three. This is the "phantom key". Consider the situation when keys 3,6 and 7 are pressed simultaneously and column 6 is scanned.

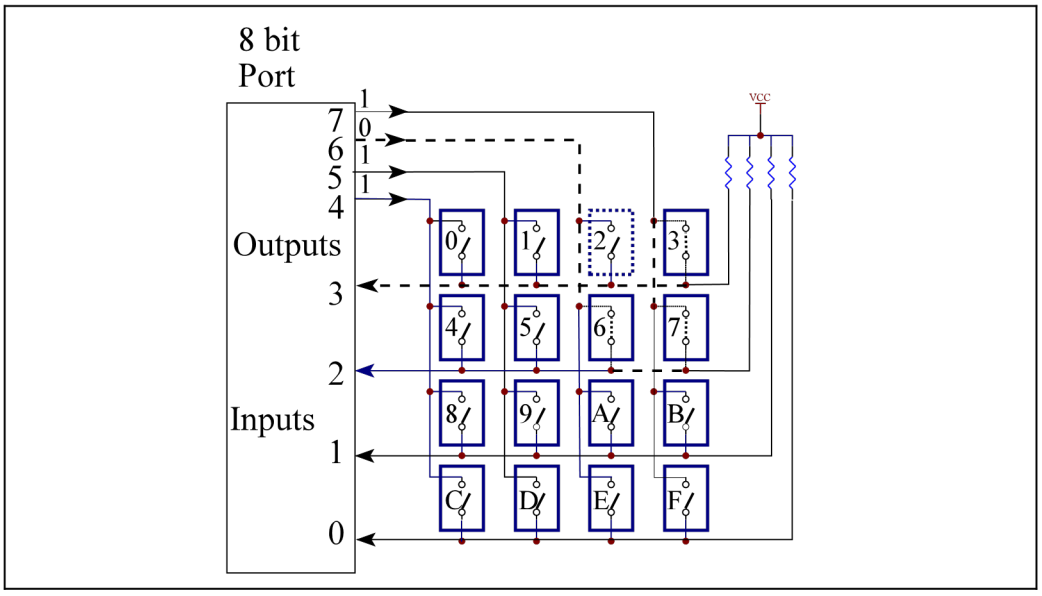


Fig 4.8 “Phantom” Key

Even though key “2" is not pressed, row 3 has a “0" value appearing there due to the flow of current in the other 3 switches. Key “2" is the phantom key. The problem can be overcome in software.

An example of a program used for keyboard scanning follows:

```
unsigned char ScanKey ();
unsigned char ProcKey ();

#define Col7Lo    0x7F                    /*  column 7 scan */
#define Col6Lo    0xBF                    /*  column 6 scan */
#define Col5Lo    0xDF                    /*  column 5 scan */
#define Col4Lo    0xEF                    /*  column 4 scan */
#define TRUE      1                       /*  neater to use! */
unsigned char ScanCode;      /* hold scan code returned     */ void
main (void)
{                          /* main entry for program      */
unsigned char i;

    while (TRUE)
    {
       i = ScanKey();
       if (i != 0xFF)  /* if key is pressed **/
                LEDPort = Bin2LED[i];   /* output to LED */


     }
}                                        /* loop forever */

/* Check for key press: if none, return 0xFF */ unsigned
char ScanKey()
{
       KbdPort = Col7Lo;               /* bit 7 low  */
       ScanCode = KbdPort;             /* Read */
       ScanCode |= 0xF0;               /* high nybble to 1 */ ScanCode
       &= Col7Lo;                     /* AND back scan value */

       if (ScanCode != Col7Lo)         /* in<>out: get key & disp */
       return ProcKey();

       KbdPort = Col6Lo;               /* bit 6 low  */
       ScanCode = KbdPort;             /* Read */
       ScanCode |= 0xF0;               /* high nybble to 1 */
       ScanCode &= Col6Lo;             /* AND back scan value */

       if (ScanCode != Col6Lo)         /* in <> out, get key */
       return ProcKey();

       KbdPort = Col5Lo;               /* bit 5 low  */
       ScanCode = KbdPort;             /* Read */
       ScanCode |= 0xF0;               /* high nybble to 1 */
       ScanCode &= Col5Lo;             /* AND back scan value */

       if (ScanCode != Col5Lo)         /* in <> out, get key */
       return ProcKey();

       KbdPort = Col4Lo;               /* bit 4 low  */
       ScanCode = KbdPort;             /* Read */
       ScanCode |= 0xF0;               /* high nybble to 1 */
       ScanCode &= Col4Lo;             /* AND back scan value */
```

```
        if (ScanCode != Col4Lo)          /* in <> out, get key  */
        return ProcKey();

        return  0xFF;                    /* no key press rtn with FF */
}   /* main */

/* Procedure here */ unsigned
char ProcKey()
{
    unsigned char i;                   /* index of scan code returned */
    for (i = 0 ; i <= 12; i++)
    if (ScanCode == ScanTable[i])              /* search in table */
       return i;                               /* exit loop if found */
    if (i == 12)
       return  0xFF;                 /* if not found, return 0xFF */
}
```

5 Liquid Crystal Displays

5.1 Introduction

Liquid Crystal Displays (LCD) are widely used as output devices. This because they consume little current as compared to LEDs and other display devices. Also they can be made physically small and have colour capabilities.

While bare LCD displays are available, it is quite troublesome to interface to one. LCD *modules* on the other hand, incorporate a graphic processor and interface electronics. In doing so modules have the added advantages of :

a) Having relatively powerful display functions and can be easily controlled and interfaced.

b) Not requiring a need to refresh the display.

Selecting a LCD module involves two basic design decisions.

1) What size and format is required to display the desired information.

2) What optical characteristics will look best in the package and attract the user to the product. This chapter mainly deals with the alphanumeric (A/N) or character type modules.

Alphanumeric modules display characters, numerals, symbols and some limited graphics. They are normally connected to a host processor through a parallel data bus, although serial interfaces are available.

Display control features such as Character Generation, Display RAM Addressing, Cursor Scrolling, Blanking, and are all included. User programmable fonts are supported.
There are various LCD modules available on the market from many manufacturers. The LCDs come in various sizes, with 1-4 lines, 16 to 40 characters per line, and 5x7 or 5x10 dot display fonts. Character height spans 9.130" (3.31 mm) to 0.500" (12.71 mm). Most formats are available in a variety of packages to meet various mounting requirements. Multi-line models offer the best value when analyzed by a "cost per character" basis. By selecting a backlight option displays are visible both day and night. Extended temperature modules are available which operate between -20 and +70 C.

For requirements of more than 4 lines or 40 characters across, select a graphic formatted

module. Graphic modules are also used when different sized characters are needed, and when special fonts such as Chinese or Arabic are required.

5.1.1 LCD Fluid Types

LCDs use a fluid dye which type determines the contrast ratio, viewing angle, and temperature range. There are 3 basic types of dye: TN (Standard type), NTN (high contrast type), and STN (premium high contrast type). Many TN and NTN models are available in extended temperature range.

5.1.2 Character LCD Controller

Software determines what, how and where data is displayed on the LCD. Most character modules use the Hitachi HD44780 or equivalent controller IC. Some of its features are:

Built-in character generator with 192 character modified ASCII character set Ability to program up to 8 user defined characters.
Bi-directional 8 or 4 bit bus interface
80 character RAM
Automatic reset on power up
Wide range of instruction functions including:
Display clear, ON/OFF, Cursor positioning, Display or cursor shift on data entry

5.2 LCD Hardware overview

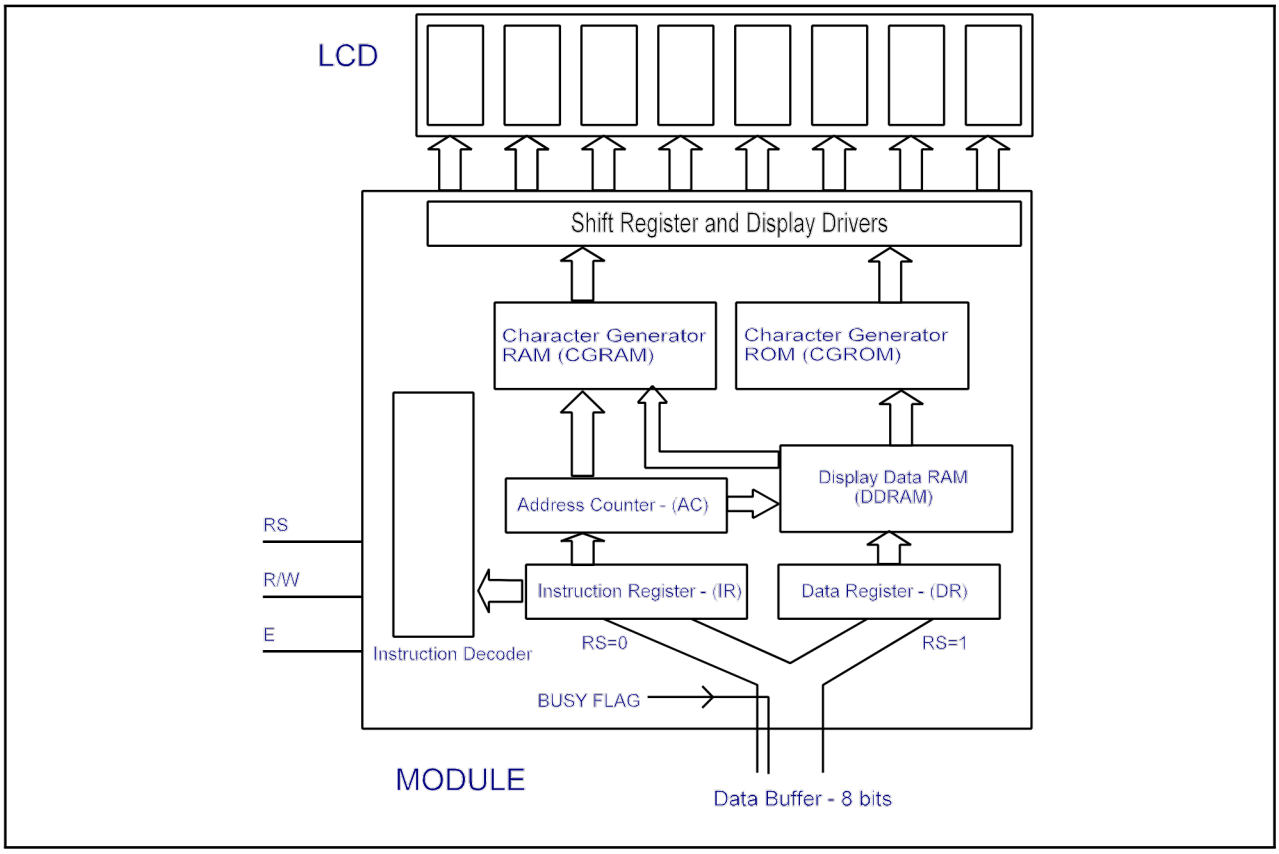


Fig 5.1 Block Diagram of LCD module

General operation

The module interfaces with the host processor through a data bus. By controlling the RS pin, data goes to either of two 8-bit registers: an instruction register (IR) or a data register (DR). The IR stores instruction codes, such as display clear and cursor shift, and address information for display data RAM (DDRAM). The IR can only be written from the host processor.

When the host processor writes to the Data Register, this register will temporarily hold the data. After that, it will be automatically transferred to the DDRAM by an internal operation.

When address information is written into the IR, data is read and then stored into the DR from DDRAM by an internal operation. Data transfer from the module is completed when the host processor reads the DR.

When reading from the module, the DR will hold data read from DDRAM. After the host processor does a read, a special feature of the module is that, data at the next address in DDRAM is automatically read. This is sent to the DR in preparation for the next read.

Data Display RAM

As implied by the name, DDRAM is used to store data to be displayed. These are 8-bit character "modified" ASCII codes. It can handle 80 x 8 bits, or 80 characters. If the actual number of characters that can be viewed is less than this, the unused DDRAM can be used as general data RAM.

Character Generator ROM/RAM

CGROM holds the actual bit patterns to be shown on the display. In the Alphanumeric modules we are considering, these patterns are constructed using a 5 by 8 bit matrix. Internal logic in the LCD module takes data from DDRAM. It then calculates where the bit pattern for that data is stored in CGROM. These patterns are shifted out to the display drivers and there to the individual LCD display elements.

The CGROM in a standard module has 160 characters in a 5x8 bit pattern format and 32 characters in 5x10 pattern.

For added flexibility, users can define up to 8 special character or symbols in a 5x8 format (for 5x10, only 4 characters). These bit patterns are stored in another type of RAM known as Character Generator RAM. Once programmed, the newly formed characters may be accessed as if they were in the "normal" CGROM. It must be reprogrammed if power to the module is interrupted. Further information on how to make use of CGRAM may be found in the LCD module databook.

Relationship between DDRAM, CGROM

To illustrate, let's say the value 62H (ASCII 'b') is sent to the Data Register. From there it goes into the DDRAM at an address specified by the Address Counter. The module logic will access the bit pattern stored in the CGROM. This is shifted out and displayed on the actual display as shown below. A '1' will turn on the pixel in a row. Note that characters are shown as a 5 by 7 dot matrix but stored as an 8 byte (8 by 8) array. A complete list of character codes and bit patterns is shown later in Figure 5.6.

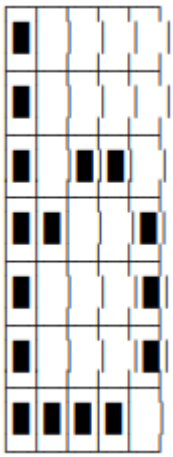
DDRAM	CGROM/CGRAM	Actual LCD
62H	00010000 00010000 00010110 00011001 00010001 00010001 00011110 00000000	

Fig 5.2 Displaying a 'b' with a 5x7 font

Busy Flag (BF)

When the busy flag is 1, the module is in the internal operation mode, and the next instruction will not be accepted. In this state, if the host processer performs a read, the busy flag is output to DB7. The next instruction must be written after ensuring that the busy flag is 0.

Address Counter (AC)

The address counter (AC) is used to access data in DDRAM or CGRAM. When a "set address" instruction is sent to the Instruction Register (IR), the address information is sent from the IR to the AC. The type of instruction determines whether DDRAM or CGRAM is addressed.

After writing to DDRAM, the AC is automatically incremented by 1. When reading from DDRAM, it is decremented by 1. The AC contents are then output to DB0 to DB6. Note that the AC register holds only 7 bits.

When addressing CGRAM, for example, to load in a user defined character, the cursor or blinking effect will appear. This effect should be ignored as the AC is pointing to a CGRAM address and thus does not affect the DDRAM.

Through the register selector (RS) signal, these two registers can be selected as shown:

RS	**R/W**	**Operation**
0	0	write to IR to start an internal operation (display clear, etc.)
0	1	read busy flag (DB7) and address counter (DB0 to DB6)
1	0	write to DR for display (DR to DDRAM)
1	1	read from DR the contents of DDRAM (DDRAM to DR)

Timing Generation Circuit

The timing generation circuit generates timing signals for the operation of internal circuits such as DDRAM and CGROM. RAM read timing for display and internal operation timing by the host processor access are generated separately to avoid interfering with each other. Therefore, when writing data to DDRAM, for example, there will be no side effects, such as flickering, in areas other than the display area.

Cursor/Blink Control Circuit

The cursor/blink control circuit generates the cursor or character blinking. The cursor or the blinking will appear with the digit located at the display data RAM (DDRAM) address set in the address counter (AC). For example, when the address counter is 08H, the cursor position is displayed at DDRAM address 08H. When a character is written to the Data Register, the ASCII character will appear at the cursor position.

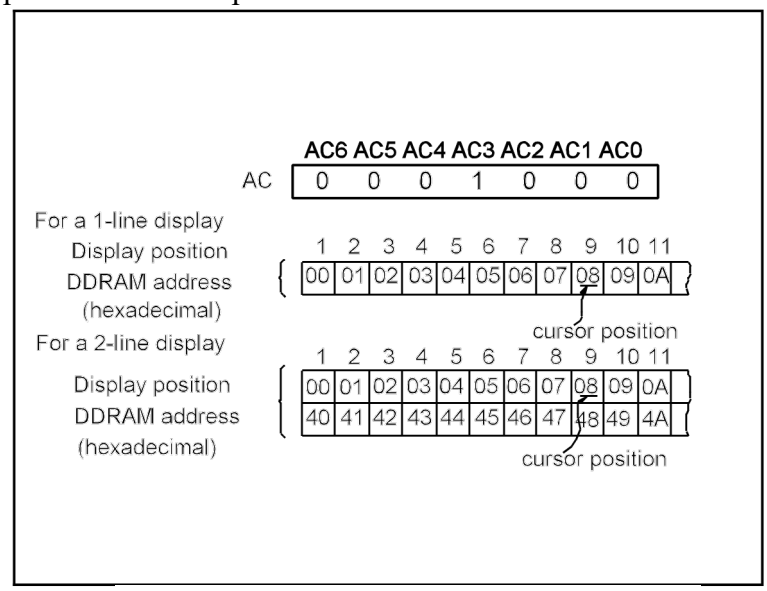


Fig 5.3 Cursor position

5.2.1 Relationship between DDRAM and Display Characters

The DDRAM has a capacity of 80 characters. If the physical display has less than 80 characters, what is on the display is a "window" on the DDRAM. The following address diagrams on the next page show DDRAM addresses as they appear after a Clear Display or Return Home instruction, or when Entry Mode Set instruction has bit S=0.

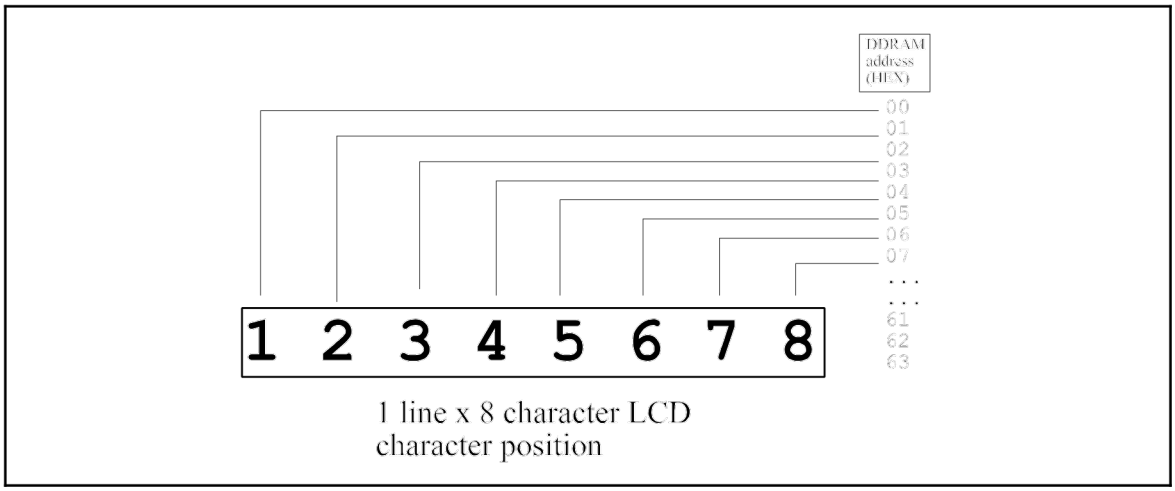


Fig 5.4 One Line Display - Character Position vs. DDRAM Address

When there are fewer than 80 display characters, the display begins at the starting position. From the diagram above, we see that DDRAM memory location 00 corresponds to character position 1.

So when we write to DDRAM address 00H, it will appear at display position 1. Note that characters in DDRAM memory locations higher than 08 are not visible for this 1 line by 8 display.

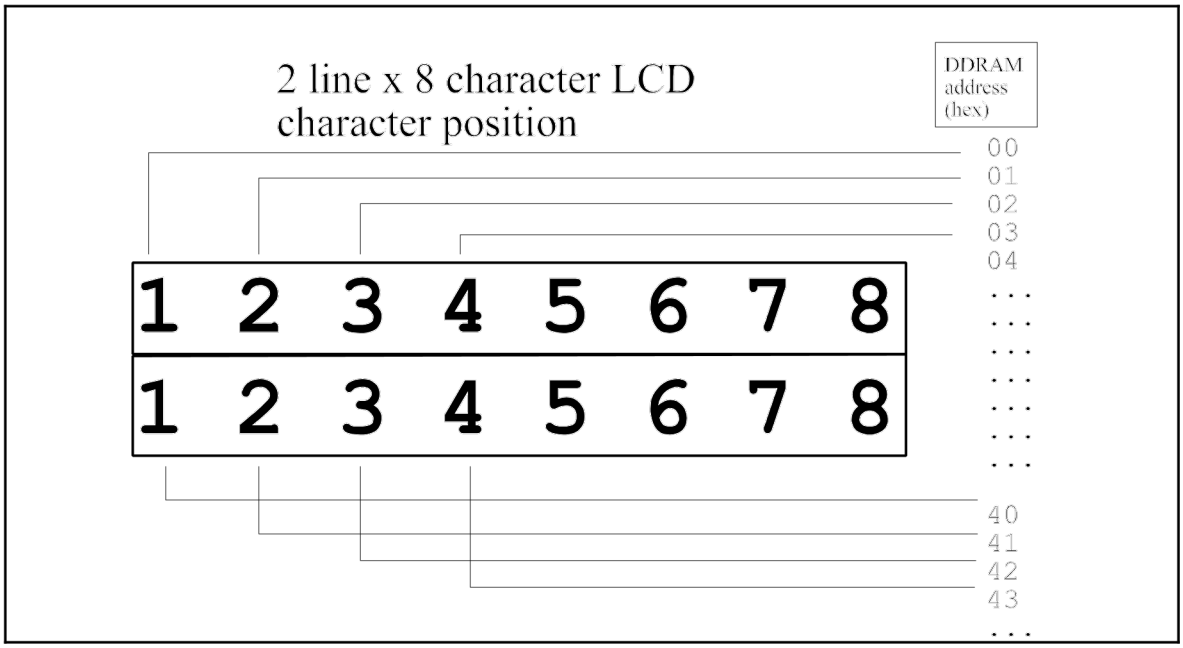


Fig 5.5 Two Line Display - Character Position vs. DDRAM Address

In a two line display, the first character on the next line is *not* "one more" than the last character on the previous line. In the 2 line by 8 character module above, what we think is the 9th character (first character on line 2) is *not* found at DDRAM address 08. This first

character on line 2 is actually at DDRAM address 40h. To put data on the second line, a Set DDRAM Address instruction must be sent.

It is possible to shift the display so that the first character does not correspond to DDRAM location 0. Thus This we can make the characters rotate through the display when data is entered. It also lets small displays, for example 2x16, to have data stored in non-visible areas of the DDRAM and have them shifted in to be viewed. This is covered in the discussion on "shift mode".

5.2.2 Character code and bit patterns

As mentioned before, typical LCD modules store the bit patterns in CGROM. They use a modified ASCII code. For example, the character 'A' is binary 0100 0001 or 41H. This is shown in the diagram below. Character codes E0H to FFH are 5x10 bit patterns.

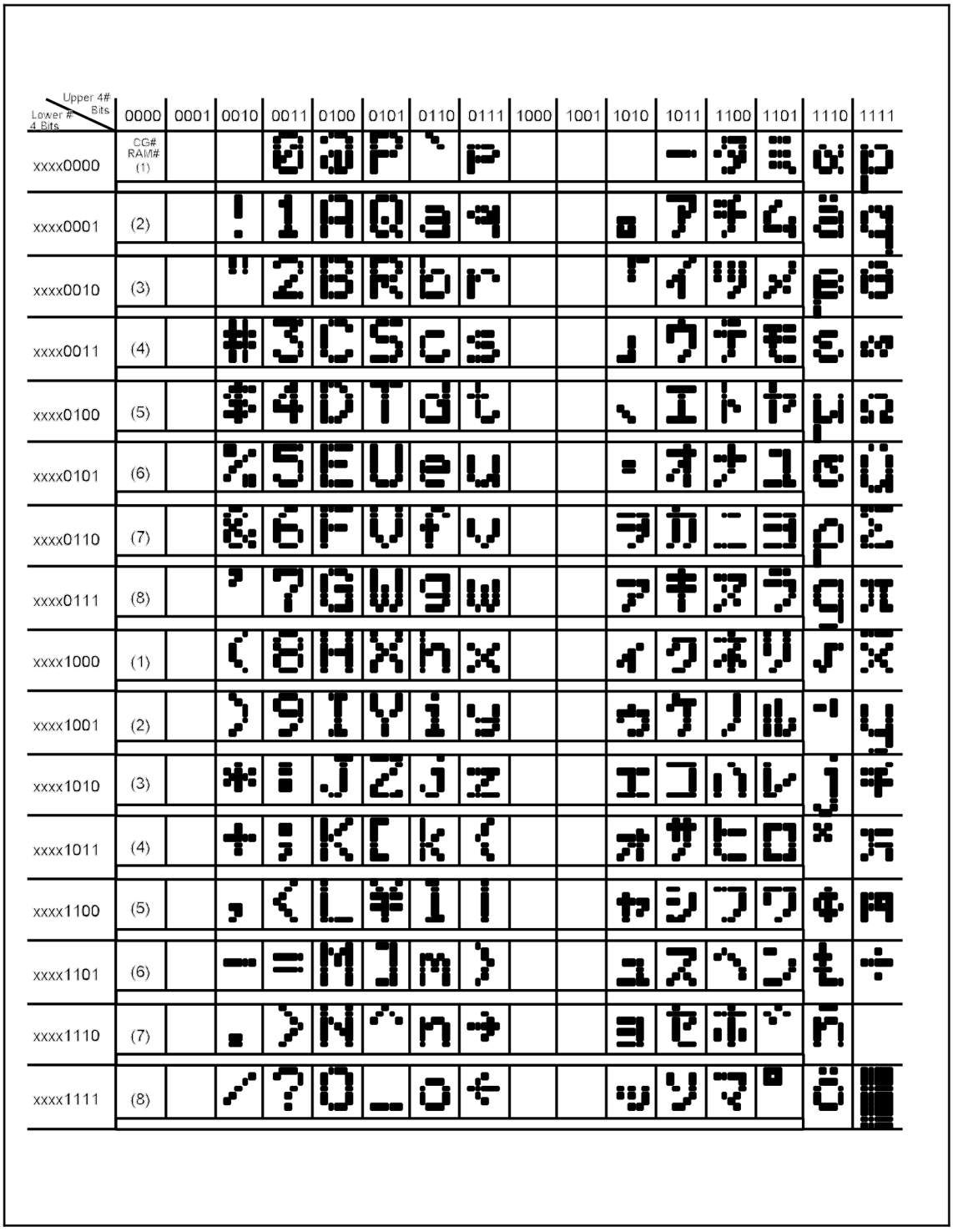


Fig 5.6 Correspondence Between Character Code and Character Pattern

5.3 LCD Software control

Only the Instruction Register (IR) and the Data Register (DR) of the LCD module is accessible to the host processor. By writing to these registers, we control how data is to be displayed on the module.

5.3.1 General Considerations - timing

An LCD module takes a relatively long time to execute instructions. On the average, this is around 40 μs. When an instruction is being processed only the Busy Flag/address read instruction can be executed. The busy flag is set to 1 and will go to logic 0 when done.

Busy Flag vs standard time delay

Most of the time, data is written to the LCD module and keeping track of its state is done by the host processor. In this case, it is not necessary to read from the LCD very often, or not at all! We can dispense with having to read the Busy Flag, and use a standard time delay of 40 μs for most instructions. In many cases, this will save the use of a buffer (or needing a bidirectional port) to accommodate the read function.

There are four categories of instructions namely:

- setting the modes of operation, such as display format, data length, etc.
- set internal addresses, both for DDRAM and CGRAM
- Perform data transfer with internal RAM
- Perform miscellaneous functions

Normally, the most commonly used instructions are those that perform data transfer to DDRAM. The autoincrement (or decrement) feature of the AC register mentioned earlier, makes it easier to program the module.

5.3.2 Default Initialisation

Before programming the LCD module, we must know its initial state when power is turned on. The following instructions are executed during the initialization period. This lasts for 15 ms after the power supply rises to 4.5V. The busy flag (BF) will be at logic one until the initialization ends when it will be at logic zero. If the rise time of the power supply meets the criteria below, the module will default to the following functions via an internal initialization routine:

1.	Clear Display	
2.	Function Set	DL=1: 8 bits interface N=0: 1 line display F=0: 5x7 dot font
3.	Display ON/OFF control	D=0: Display OFF C=0: Cursor OFF B=0: Blink OFF
4.	Entry Mode Set	I/O=1: +1 increment
5.	DDRAM is selected	

If the power is not applied successfully, the initialisation routines will fail. Also, if different parameters are required such as for a 2 line display are required, the host processor has to perform the initializing. We will look at some examples later.

After initialising, we should have a clear display with a flashing cursor in the upper left position. The cursor will then increment to the right with each DDRAM write command.

5.3.3 INSTRUCTION TABLE

The following is a summary of the instructions that can be sent to the LCD module and their typical timings as well. Some instructions can perform several operations at one time depending on the setting of individual bits. These bits are formed into a data byte which is sent to the Instruction Register.

Instruction	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution Time**
Clear Display	0	0	0	0	0	0	0	0	0	1	Clears Display and returns cursor to the Home Position (Address 00)	80us = 1.64ms
Return Home	0	0	0	0	0	0	0	0	1	*	Returns cursor to Home Position. Returns shifted display to original position. Does not clear display	40us = 1.6ms
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S	Sets DD RAM counter to increment or decrement (I/D) Specifies cursor or display shift during to Data Read or Write (S)	40us
Display ON/OFF Control	0	0	0	0	0	0	1	D	C	B	Sets Display ON/OFF (D), cursor ON/OFF (C), and blink character at cursor position	40us
Cursor or Display Shift	0	0	0	0	0	1	S/C	R/L	*	*	Moves cursor or shifts the display w/o changing DD RAM contents	40us
Function Set	0	0	0	0	1	DL	N	F	*	*	Sets data bus length (DL), # of display lines (N), and character font (F)	40us
Set CG RAM Address	0	0	0	1	A_{CG}						Sets CG RAM address. CG RAM data is sent and received after this instruction	40us
Set DD RAM Address	0	0	1	A_{DD}							Sets DD RAM address. DD RAM data is sent and received after this instruction	40us
Read Busy Flag & Address	0	1	BF	AC							Reads Busy Flag (BF) and address counter contents	1us
SIZE=2>Write Data from DD or CG RAM	1	0	Write Data								Writes data to DD or CG RAM and increments or decrements address counter (AC)	40us
Read Data from DD or CGRAM	1	1	Read Data								Reads data from DD or CG RAM and increments or decrements address counter (AC)	40us
I/D=1: Increment S=1: Display Shift on data entry S/C=1: Display Shift (RAM unchanged) R/L=1: Shift to the Right DL=1: 8 bits N=1: 2 Lines F=1: 5x10 Dot Font D=1: Display ON C=1: Cursor ON B=1: Blink ON BF=1: Cannot accept instruction				I/D=0: Decrements S=0: Cursor Shift on data entry S/C=0: Cursor Shift (RAM unchanged) R/L=0: Shift to the Left DL=0: 4 bits N=0: 1 Line F=0: 5x7 Dot Font D=0: Display OFF C=0: Cursor OFF B=0: Blink OFF BF=0: Can accept instruction							Definitions: DD RAM: Display data RAM CG RAM: Character generator RAM A_{CG}: CG RAM Address A_{DD}: DD RAM Address(Cursor Address) AC: Address Counter used for both DD and CG RAM Address	Execution Time changes when Frequency changes per the following example: If F_{CP} or f_{osc} is 27 KHz 40us x 250/270 = 37us

* Don't Care **(when F_{cp} or f_{osc} is 250KHz)

5.3.4 Special operation modes

Four bit operation

We have been describing how data can be sent as 8 bit bytes. LCD modules are able to receive this byte as two 4 bit nybbles. In many cases this can save an extra 8 bit latch, not to mention decoding resources. This slight increase in complexity can be hidden in the host processor's output routine. The main user program then calls this routine with the standard 8 bit byte, leaving the two 4 bit transfer to be handled by the routine. We also need to initialise the module into 4 bit mode.

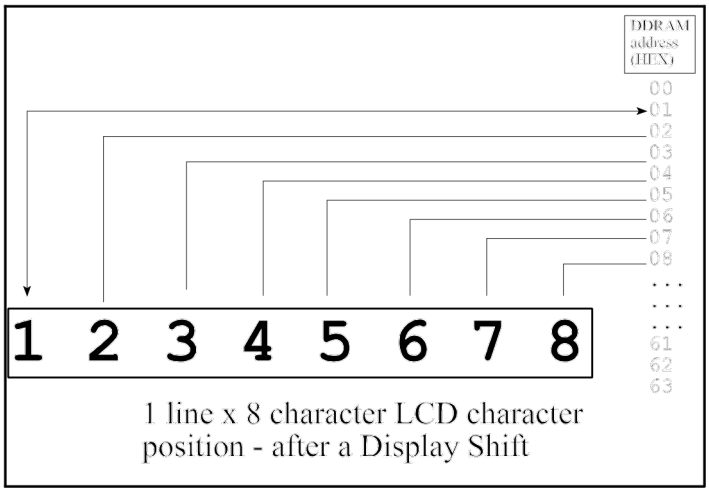


Fig 5.7 Display shift operation in LCD

Display Shift

DDRAM can be used for displays such as for advertising when combined with the display shift operation. When instruction S=1, the display is shifted. What is displayed depends on the Entry Mode Set instruction. The display shift operation changes only the display position with DDRAM contents unchanged. After a Display Shift Right command, we can see that DDRAM address 01 now corresponds to display character 01 and the character at DDRAM address 08 is now visible on this 1 line by 8 display. Similarly, a Display Shift Left will bring the last character of DDRAM into view. For a two line display, both lines will be shifted in the same way.

5.4 Further initialisation considerations

With a proper understanding of the instructions available, we can look in greater detail at the initialisation routines. Remember, this is only necessary when the default initialisation is not sufficient. These are placed in the chapter appendix.

Sample display program

As a summary, the following shows the commands sent to set up a 2 line LCD module with 5x7 font, 8 bit data transfer, display address increment and shifting the cursor to the right when writing. This will also display a simple message.

Step #	Instruction									Display	Operation
	RS	R/_W	DB7	DB6	DB5	DB4	DB3	DB2	DB1 DB0		
1	Power supply on (the HD44780 is initialized by the internal reset circuit)										Initialized, no display
2	Function set		0	0	1	1	1	0	x x		Sets to 8-bit operation and select 2-line, 5x7 dot font. (after this, no. of lines and character fonts cannot be changed.)
3	Display on off control		0	0	0	0	1	1	1 0	_	Turns on display and cursor. Entire display is in space mode because of initialization.
4	Entry mode set		0	0	0	0	0	1	1 0	_	Sets mode to increment the address by one and to shift the cursor to right at the time of write to the DD/CGRAM.
5	Set 00H to DDRAM address		1	0	0	0	0	0	0 0	_	Start from the first character of the first line
6	Write data to DDRAM		1	0	0	1	0	0	0	H_	Writes “H”. The cursor is incremented by one and shifted to the right
7	Write data		1	0	0	1	0	0	1	HI_	Writes “I”.
8	Set DDRAM address to 2nd line		1	1	0	0	0	0	0 0	HI _	Cursor goes to second line
9	Writes data		1	0	0	1	1	1	1	HI O_	Writes “O”
10	Write data		1	0	0	1	0	1	1	HI OK_	Writes “K”
11	Write data		1	0	0	0	0	0	0	HI OK _	Writes space

LCD 8 Bit Operation, 8-Digit x 2-line Display Example

5.5 Other Display Control Considerations

So far, we have see how to transfer characters from a host processor to the DDRAM of a LCD module so they can be displayed. We also positioned the cursor for better control of the message. Now if there are a lot of messages to be displayed, there may be difficulty in keeping track of what is displayed. In this section we look at these considerations.

5.5.1 Direct DDRAM manipulation

This is the method we have covered in the earlier example, where for each message, the cursor is positioned first (by setting the Address Counter). This is fast and efficient as we only display what is needed. If there is a lot of screen updates, after a while it is hard to keep track of what needs to be displayed and erased, to make sure we don't leave behind previous messages. This makes the display code hard to debug and understand.

5.5.2 Host buffer construction

In this method, we construct a memory buffer on the host processor which will reflect what is to be displayed on the LCD. This buffer is typically implemented as a character string, these size being equal to the number of characters on the LCD. There may be redundant updates of messages. But the advantages are:

i) assuming the host processor has the necessary debug support, we can examine the buffer before it is sent to the LCD.
ii) that existing data conversion routines in the supplied libraries can be used. For example, floating point and hexadecimal can be easily displayed.

We will illustrate this with C string operations, which are normally accessed by including the `strings.h` file in the host program. The buffer should be an `unsigned char` block of memory. This being so, we can use `sprintf` and the same format specifier as `printf`, to format our strings. In addition, we can use operations like `strcpy` (string copy) and `strcat` (join strings) to manipulate the strings further.

A further consideration is that although a `sprintf` command can have a message and a variable as in :

```
sprintf(LCDStr,"Subject:ET%d-OK",test);
```

It is more advisable to keep strings in a separate place, perhaps even in a header file, where they can be more easily accessed rather than searching the C program for where the message is. For example:

```
sprintf(LCDStr,"%s %d",message, test);
```

5.6 LCD Hardware Design

Here we consider the power supply, lighting control and the logic signals.

5.6.1 POWER SUPPLY REQUIREMENTS

Power Supply to Module Connection

Modules require +5V at 1 to 10 milliamps. Extended temperature and some high contrast modules require -5V, also at low current. Inexpensive ICs convert +5V to -5V efficiently. If the display has backlighting, required power must also be budgeted.

A module's logic circuits have 3 connections to the power supply: VDD (+5VDC); VSS (Ground); and V_O, viewing angle adjustment, sometimes called contrast or bias control. The diagrams show typical connections. Contrast can also be controlled digitally with a digital potentiometer or a Digital Analog Converter.

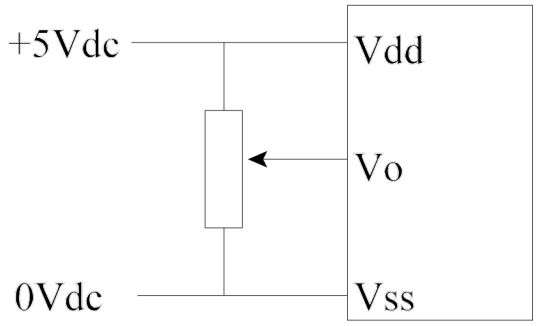


Fig 5.8 Power connections to LCD module

5.6.2 Logic connections

As we have seen, the LCD module appears as two 8 bit registers to a microprocessor system. Additionally, there are three control pins, making a total of 11 pins or two registers. Generally, interfacing the module to an existing system involves:

a) applying the proper viewing angle voltage to the display's V0 pin.
b) connecting the module to the host processor - directly to a port, or through buffers and latches
c) developing a strobe signal for the "E" signal
d) applying the appropriate "RS" and "R/W" signals to modules

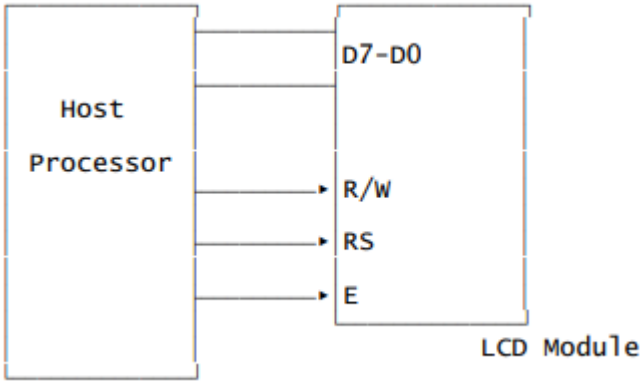


Fig 5.9 LCD 8-bit interfacing diagram

The 8 bit mode is more straightforward and the software control is simpler. It needs 8 bits for data and 3 control signals. The timing is shown below. We need two 8 bit devices to interface to it.

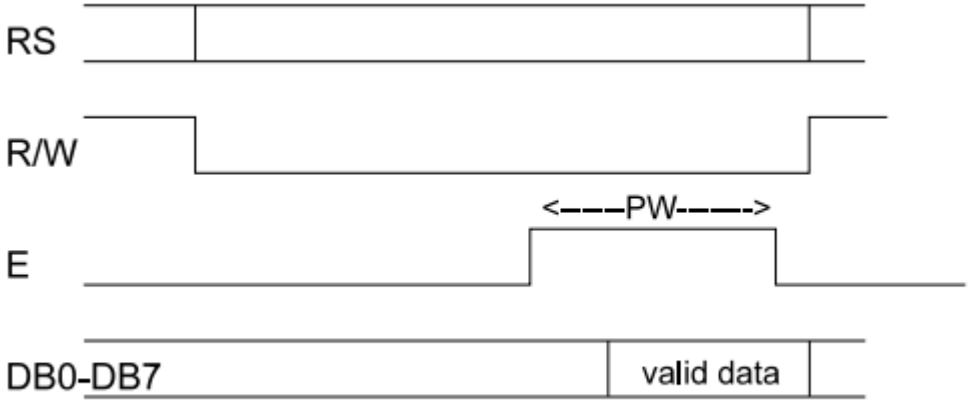


Fig 5.9 Timing diagram of LCD module

Timing Aspects of Alphanumeric LCDs

Alphanumeric LCDs are classified as slow peripherals. The Enable ("E") signal is the key signal line. This signal clocks the data and control signals into the LCD module. The E signal must be a clean, positive going digital strobe, which is active while data and control information are stable and true.

The two control lines, RS and R/W, must be set up before the activation, or rise, of E. These signals must remain stable for 10ns after the fall of "E".

A likely choice for host connections are: RD and WR strobes (or the R/W line) and to connect the E pin as a “chip select” line. But the E strobe must be 450ns wide (PW) minimum. Most modern processors operate at higher speeds than this and it is not practical to slow down the host for this purpose. Often we latch both the data and control information, toggling the E signal in this way. Of course if a port is free, this may be used as well.

When a parallel port supplies RS, R/W and E, all these lines should not change together. This would result if a single instruction was employed and would surely violate the set-up requirement. Instead a second instruction must independently set the E bit high, after the other lines are set.

Another option is to tie the R/$\overline{W}$ and RS pins to the host's address lines which is set up earlier in the machine cycle. The data bus must set-up 195nS prior to the fall of "E". These lines must hold for at least 10nS after E falls. Most host strobes should meet these requirements without difficulty.

Normally E strobes would be approximately 40 microseconds apart - which is the maximum display throughput. (See Instruction Table for complete list of execution times).

Four bit connection

An important advantage to using a four bit transfer mode is that it only needs 4 data lines and 3 control lines. So one 8 bit port will be enough. This reduces the number of 8 bit devices used. There is only a slight increase in software complexity. The following shows two ways of connecting up. One is directly to a port, the other through a latch.

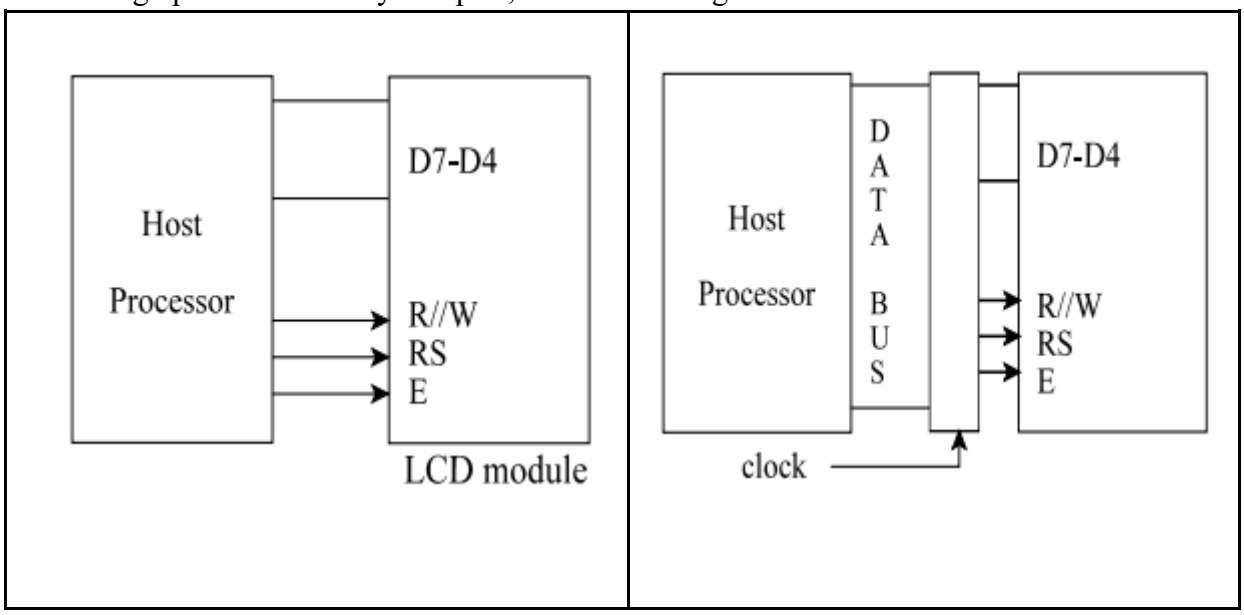
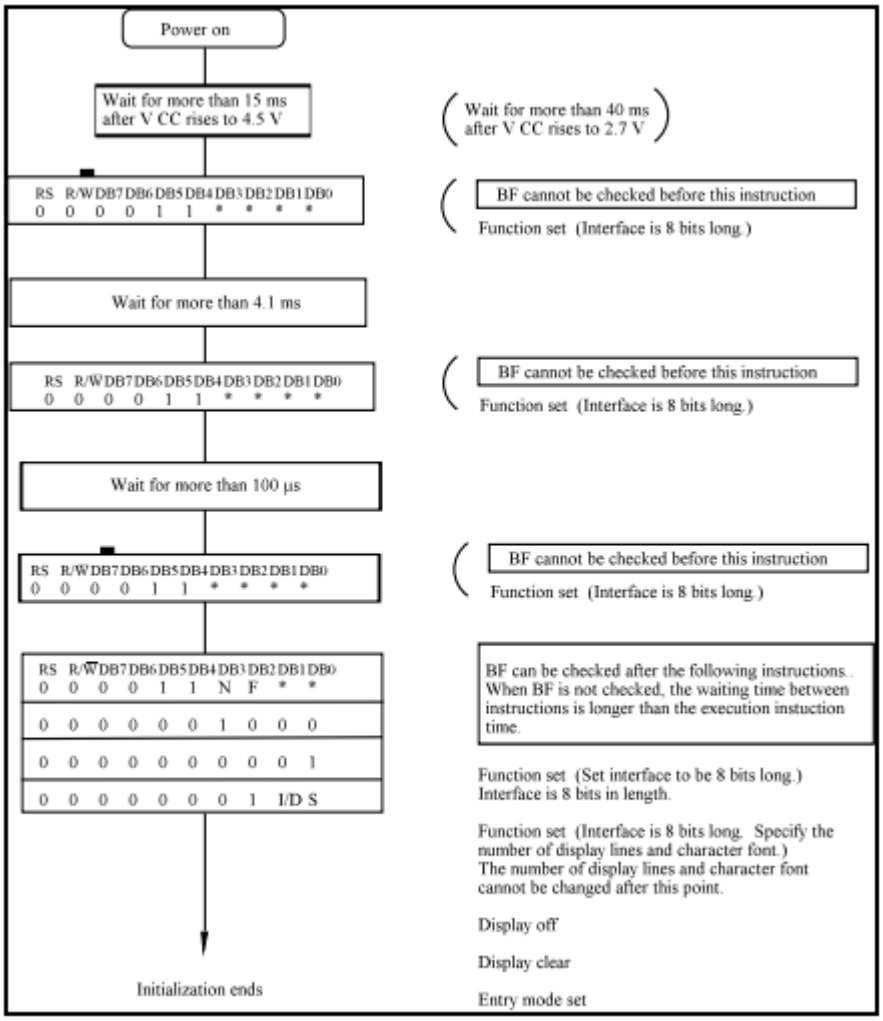


Fig 5.10 Four bit data transfer Port/Latch Interface

The data transfer between the module and the host processor is completed after the 4-bit data has been transferred twice. The four high order bits (DB4 to DB7) are transferred first, then the four low order bits (operation, DB0 to DB3). If the busy flag is used, it must be checked after the 4-bit data has been transferred twice. This being one instruction. Two more 4-bit operations then transfer the busy flag and address counter data.

8 bit initialisation



In summary, there should be a 15ms delay after power up before initialization begins. Then there are 4.1ms and a 100 µs delays where the "3X" codes are sent. The following are some sample initialisation codes for various configurations (all in hexadecimal)

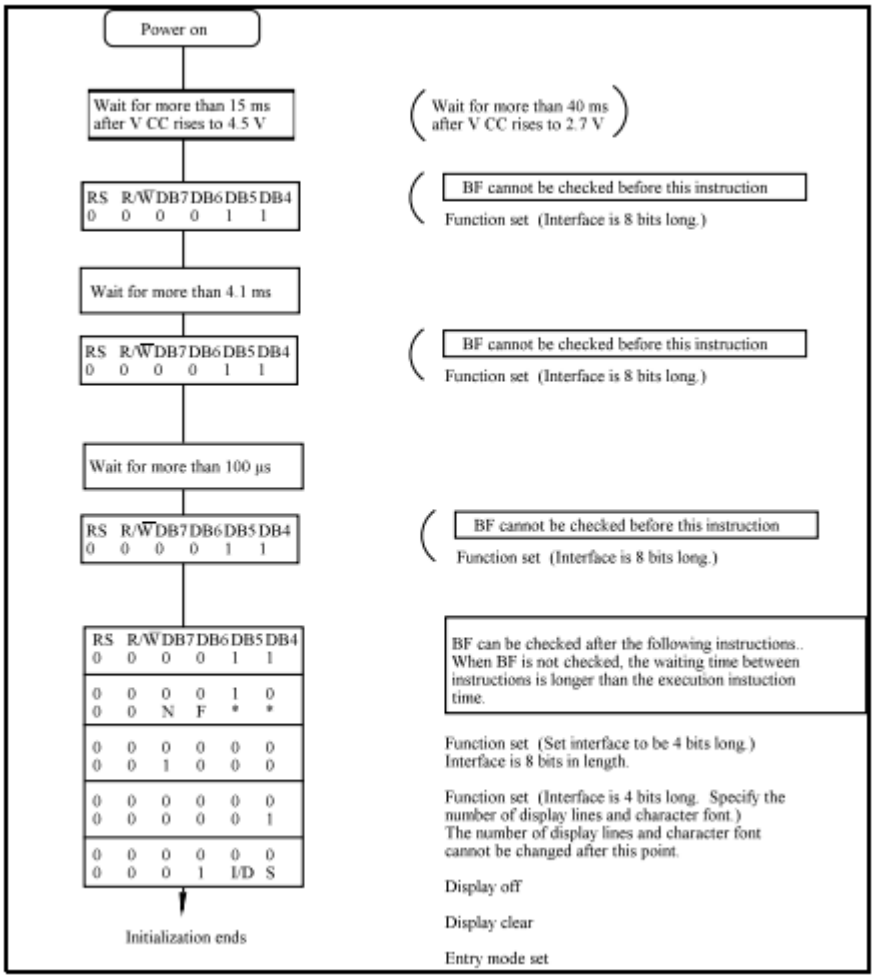
1 line display with 5x7 font 30, 30, 30, 30, 08, 01, 06

1 line display with 5x10 font 34, 34, 34, 34, 08 ,01, 06

2 line display with 5x7 font: 38, 38, 38, 38, 08, 01, 06

Of course, a 0E code should be sent to turn on the display!

Initialization for 4-bit operation



The modules will operate from a 4-bit wide data bus. Data is transferred over data lines D7-D4. D3-D0 may float. An 8-bit hex code is sent one nybble at a time, the higher nybble first. The function set in the initialization routine must change to accommodate this mode. A recommended routine follows

2 line display with 5x7 font 28, 28, 06, 0E, 01

6 Stepper Motors

6.1 Introduction

Stepper motors are useful in applications where a high degree of positional control is required. This is true in computer and instrumentation applications, where they are used in printers, robots, floppy and hard disk drives, for example. This is in contrast to conventional DC or AC motors which rotate continuously. Stepper motors rotate to move a load to a given position, then stop. In the case of the floppy drive, a motor may be commanded to rotate a given number of steps to position the magnetic head. The head will then remain stationary, reading the data above that particular track. It is not possible for many types of motors to hold a load in position when it stops.

6.1.1 Ease of application

These motors are particularly useful in computer applications. This is because by just issuing a series of pulses, the motor will rotate a fixed number of steps. By changing the order of application, the motor can be made to reverse direction. To move at a given speed, we issue the pulses at a given rate. In the same way acceleration may be controlled as well. In summary, most of the common tasks of motion control can be accomplished in software. This reduces hardware costs, which can be high for precision mechanical applications.

6.1.2 Types of Stepper Motors

Three basic types of stepper motors are in common use, namely the Permanent magnet (PM), Variable reluctance (VR) and Hybrid types. We will look at the PM type.

6.1.3 Characteristics

As in other types of motors, there is a rotating part called the rotor. PM motors have a permanent magnet on the rotor. The rotor resides in a casing which physically surrounds it. In the casing there are several coils of wire called the stator. The user drives the rotor by sending current into the stator in a defined sequence. Stepper motors are classified by physical size, torque, winding resistance and step size. The size, torque and resistance determine the load to be moved and will not be considered here.

Step size

The step size is the amount the motor rotates in one step. This is purely a function of the mechanical construction of the motor. One stepper motor could have a step size of 1.8 degrees, another of a step angle of 18 degrees and so on.

Phase

This is the number of sets of windings on the stator. The most common type of motors use four phases. This will also determine the number of wires required to drive the motor. Another way of looking at the phase is the number of electrical signals which have to be applied to the motor for one electrical cycle.

Relation between phase and step size

The relationship between a phase and the actual distance a motor moves depends on the motor construction and how the motor is electrically connected. For example, switching the current from one phase to the next results in the motor moving one step.

6.2 Basic motor control

Here we will look at how a stepper motor moves a series of steps. The angle moved in a step depends on the actual physical construction of a motor. Motors commonly used in a floppy drive have step angles of 1.8 degrees. Looking at a motor with such a small angle is difficult. For convenience, we will consider a motor with a step angle of 90 degrees. In reality, a motor with such a large angle is not common. We will also assume it has four coils on its stator. This corresponds to four phases - A, B, C, D and a permanent magnet rotor with a single magnet.

Let phase A be activated first as a SOUTH pole. The rotor will align itself so the poles attract. After this, phase A is turned off and phase B is energized, as shown. The rotor will turn by 90 degrees. The magnetic force produced by the stator coils pulls the rotor to align with it.

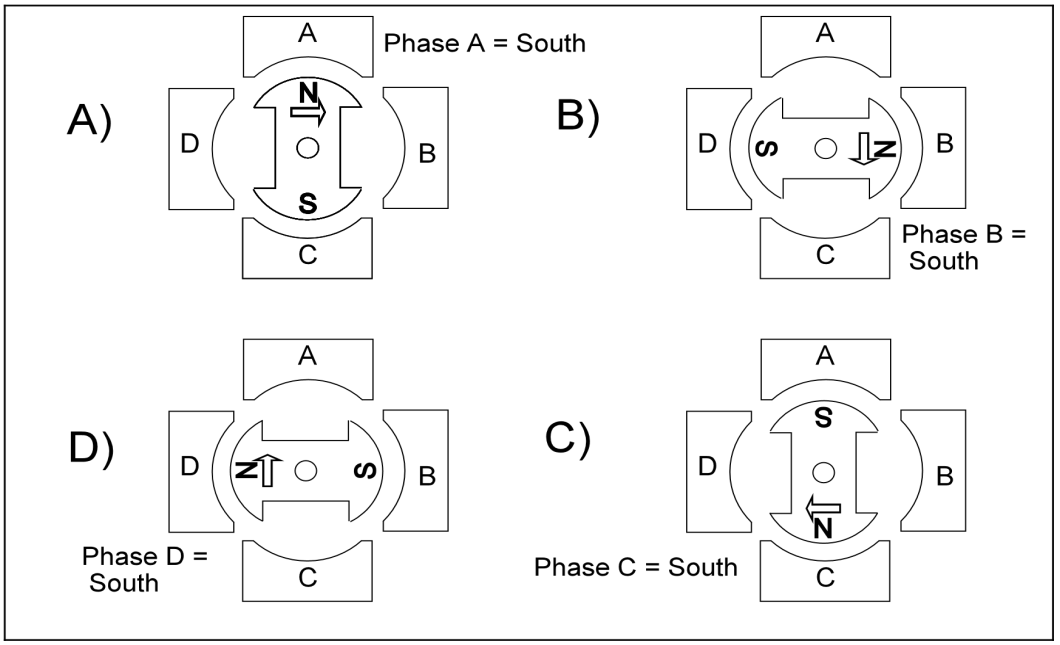


Fig 6.1 Stepper motor movement

Now phases A and B are turned off, and phase C turned on. The rotor moves another 90 degrees and so on for phase D.

Note that in this case, turning on one phase moves the motor by 90 degrees. This is not always the case

Now a microcontroller can control the current in the phases easily, by sending binary valued voltages to it. By doing so, it controls the position. By timing the signals, it will control the speed and acceleration.

The following is a schematic of a typical stepper motor circuit.

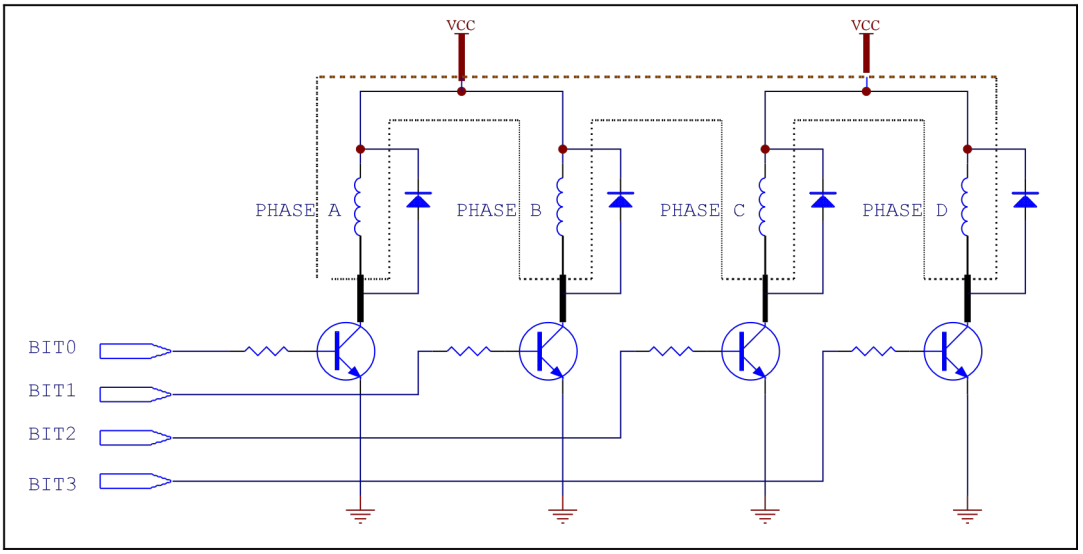


Fig 6.2 Schematic of a four phase stepper motor circuit

In this circuit, the lowest four bits of a port are used to control each phase. The motor itself is represented by the dotted lines. It is connected to the rest of the circuit by the thick lines. Altogether these represent six wires. Fly back diodes are used to redirect the current back to the power supply when the transistors turn off. Note that two coils are connected to the supply voltage. This fact may be used if the wiring of a stepper motor is not given.

6.3 Standard methods of position control

There are several ways to improve on motor positioning. We can improve on the torque, or turning force applied to the rotor. This can be done by increasing the current to the coils. We will consider only computer control here.

6.3.1 Full step

As described earlier, the motor moves 90 degrees per step as we activate the phases. This is called full stepping, where the rotor moves the entire step it was designed for.

Another way is by turning on two coils at the same time. As shown, if we turn on phases A and B at the same time, the rotor will rest at a position halfway between the two. Although twice the current flows, torque increases 1.414 times only. Then phases B and C are turned on and the motor moves another 90 degrees.

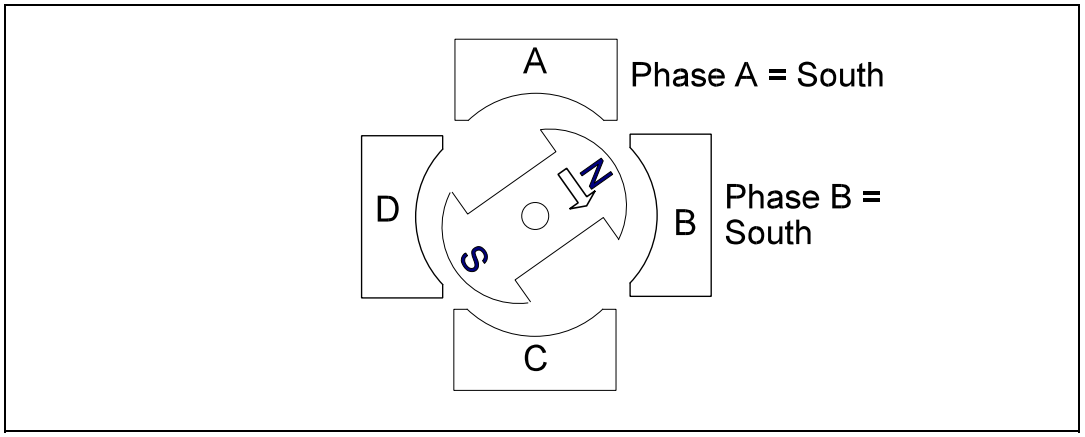


Fig 6.3 Full step - halfway between phases

We can control the movement and torque produced both by activating the proper phases and by the motor electrical connection.

Unipolar drive

So far, we have connected up the stator coils so current flow only one way through the transistor. This is called Unipolar Drive.

Bipolar drive

This works by controlling a phase with up to four transistors and diodes so current can flow both ways into a phase. For example, Phase A is activated as a South pole and later it is activated as a North pole. Some motors which have only four windings, without a common power connection, have to use this drive arrangement. Also, this makes more efficient use of electrical power. When a motor coil is turned off, the diodes conduct. This allows current to flow back to the power supply. In effect, energy is returned to the supply.

6.3.2 Microprocessor control

With the circuit above, all a microprocessor has to do it to output the required four bits to make the motor move. The table below represents the phases to be turned on to make the motor rotate one revolution, with the hex value to be output to the port above. Remember for this example the lowest four bits are connected as in the schematic shown.

Step	Phase D	Phase C	Phase B	Phase A	Value
1	0	0	0	1	1
2	0	0	1	0	2
3	0	1	0	0	4
4	1	0	0	0	8
1	0	0	0	1	1

Table 6.1 Full-Step Sequence for Clockwise Rotation.

6.3.3 Half step / other steps

If we vary the phases so that A turns on, then A and B, then B and so on, we can make the motor move in 45 degree steps. This is called half stepping and effectively increases the resolution of the motor. The sequence of phase activation to move a complete revolution in half stepping mode is shown below.

Step	Phase D	Phase C	Phase B	Phase A	Value
1	0	0	0	1	1
2	0	0	1	1	3
3	0	0	1	0	2
4	0	1	1	0	6
5	0	1	0	0	4
6	1	1	0	0	C
7	1	0	0	0	8
8	1	0	0	1	9
1	0	0	0	1	1

Table 6.2 Half-Step Sequence for Clockwise Rotation.

In full and half step movements, the full current is applied to the stator coils. If we send a fraction of the current, the rotor will come to a position in between the two phases. This technique is called microstepping.

If we activate the phases in the opposite sequence - for example D,C, B, A in full stepping, we will make the motor move in reverse.

6.4 Speed control

Although a stepper motor performs positioning duties well, it can move loads as well. It is not possible to drive a stepper motor at an arbitrary speed and acceleration. At high speeds,

the back EMF generated by the moving rotor reduces the voltage available to drive current through the motor coils. Some areas of concern in controlling movement are discussed.

Loss of synchronisation

If the rate at which the coils are energized is too high, the motor may not rotate in synchronism with the pulse rate, and may even stop moving. For example, if the microcontroller sends out pulses at 10 microseconds intervals, we expect the motor to move at a rate of 1/10 microseconds or 10,000 pulses per second. If the motor is not capable of this, it may stall - that is, stop moving or it will move at a lower rate.

If we stop sending pulses, we expect the motor to stop immediately. Again, if the speed is too high, the motor and load may continue moving even though the pulses have stopped.

Resonance

At other speeds, the motor may vibrate or even rotate in reverse all by itself. These effects are known as resonance and are due to the fact that the stepper motor is a very nonlinear device. The interaction of the various elements of the motor and its driving circuit may cause such effects to occur.

6.4.1 Torque speed curve

An important characteristic for a stepper motor in motion control is the torque speed curve. For a given motor, and a given phase voltage, the motor is given a load which is proportional to the amount of torque it can produce. A low stepping rate is applied. The motor should not lose synchronisation. When the pulses stop, the motor should stop immediately. The speed is now increased by a small amount and the same check for motor synchronism performed. This goes on until the motor just loses synchronism.

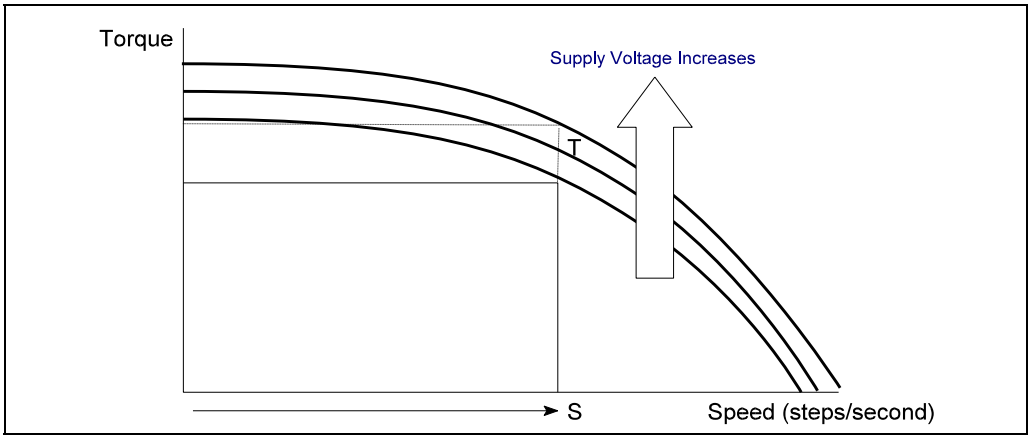


Fig 6.4 Torque Speed curve

If we look at the torque speed curve, at point T, what this means is that to produce a torque of T, the motor may be driven at any speed up to S steps per second for a given supply

voltage. The motor must remain in synchronism to the pulse rate. If we increase the supply voltage the torque speed curve shows that there is more torque for the same speed.

At high stepping rates, the rotor produces higher back EMF. This acts against the applied voltage thus reducing the motor current. As a result the torque produced will drop. Consequently if we want to move a load at a high stepping rate, we need to start slowly and gradually increase the rate, so the motor does not lose synchronisation.

One common difficulty is to determine what torque is needed to drive a certain load. Also, if the system is not properly constructed mechanically, the load will not be even.
In this case a trial and error method may be needed.

6.5 Other considerations

In stepper motor movement applications, the motor will be accelerated to a constant speed, which is also called the slewing speed. Before coming to a stop it will be decelerated. At all times the timing routine is used. These are all processor intensive activities. For situations where many motors must be controlled simultaneously additional pieces of hardware may be used. These are translators and indexers. These may be single chip devices, or a printed circuit board depending on the application.

Translator

When interfacing to this piece of hardware, the motor still has to send out steps. It this way the motor still controls the speed. But the direction of travel, whether to use half or full stepping is done by setting some pins.

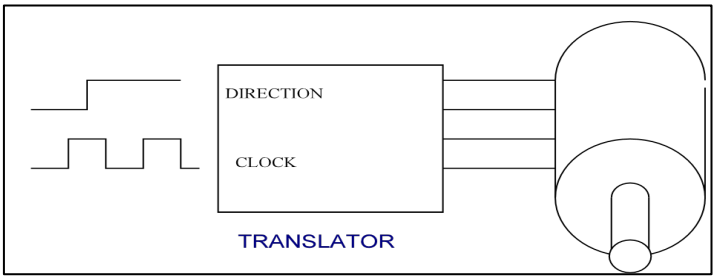


Fig 6.5 Translator block diagram

Indexer

This hardware accepts commands from the processor specifying the start/stop, acceleration and deceleration rates, slew rate, and the final position. It then generates the required stepping rates to achieve the speed profile and informs the processor when the final position has been reached.

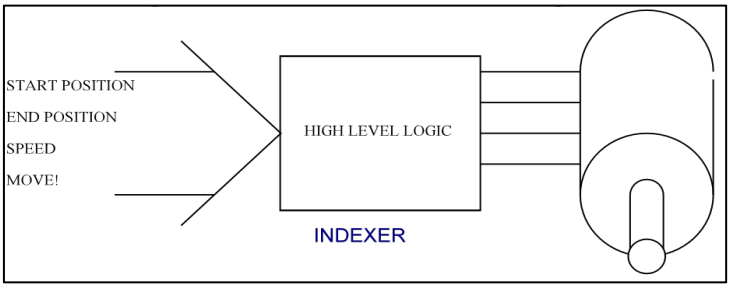


Fig 6.6 Indexer block diagram

This may be done by using an interrupt or by setting a bit in a status register. As can be seen, an indexer has a lot of functions and may include a translator circuit as part of it.

6.6 Sample application

A stepper motor rotates 1.8 degree per step. What step rate is needed to make it rotate at 10 revolutions per minute (rpm) ? Assume that the program execution time is negligible. You also have access to a routine called Delay1ms which provides a 1 millisecond delay. Use the motor in FULL step mode.

Calculation:
We need to convert from rpm to steps per second.

10 rpm = 360 degrees/minute * 10

This is equal to 3600 / 60 degrees/second
Since one step is 1.8 degrees, we need to issue 60 / 1.8 = 33.33 steps per second

Or 1/33.33 seconds per step = 0.03 which is 30 milliseconds per step.

Use the lookup table for the full step motion.
The following is a sample C program that moves the motor in halfsteps.

```
#define       SMPort        0x335
#define       NumSteps      200
#define       PtableLen     8

unsigned char Ptable []={0x01,0x02,0x04,0x08,0x03,0x06,0x0c,0x09};

void main (void)
       {
       int i, j;
       i=0;
       for (j=NumSteps;j>0;j--)
              {
               _outp (SMPort,Ptable [i]);       /* output to port */
              Sleep (300);                      /* step rate */
              i++;
              if (i>=PtableLen) i=0;
              }
       }
```

7 Digital-to-Analog, Analog-to-Digital Interfacing

7.1 Introduction

In any electronic system, two basic types of signals can be generated or measured. They are digital signals which have discrete values, and analog signals which are continuously variable Some examples are pressure, height sound and so on.

In many of the real world systems studied by scientists and engineers the parameters are analog quantities, and when electronic measurement techniques are used, data is derived in analog form from a transducer. It is possible to process, and store analog data using a purely analog system however, this may lead to difficulties in reading and recording.

Digital electronic systems can manipulate and store large amounts of data. The advent of low cost digital microprocessor systems has drastically reduced the cost of implementing digital data processing. In fact, many high integration microprocessors have analog to digital converters built into them. But for the purpose of this course, we are concerned only with devices external to the processor.

Before a digital system can be used, the analog measurements made must first be converted into digital form.

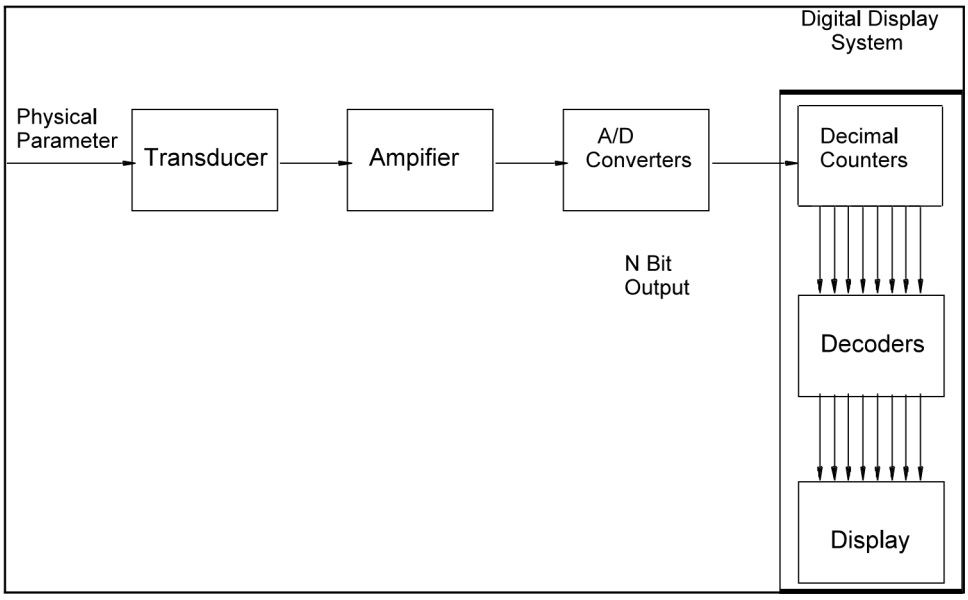


Fig 7.1 A digital instrumentation system

There are some areas where analog techniques are more useful, eg in the presentation of data where a peak or a null is to be determined.

7.2 Digital-to-analog Conversion Techniques

A digital-to-analog converter converts a digital value into a analog voltage or current by way of a resistive network. The input could be in binary or BCD, and the output will be a DC voltage or current.

Theoretically, we want the digital value be directly proportional to the to the analog output. For example, say an 8 bit DAC has a value of 255_{10} or FFH input to it. This causes an output voltage of 5 volts. An input of 127_{10} or 7FH should cause it to output 2.5 V.

A typical DAC is shown in Figure 7.2 consisting of:

a. a resistor network
b. current/voltage switches
c. a reference voltage source
d. an output amplifier for converting current to voltage.

Note that the converter does not know the analog value being output. This is controlled entirely by external hardware and essentially is just a scaling factor.

A more detailed description of a D/A converter can be found in the chapter appendix.

The next diagram shows how a voltage mode R-2R converter may be realised in practice. The accuracy of the DAC is dependent on the resistor network. We need to buffer the inputs and outputs, and provide a stable voltage reference. For more demanding applications, an integrated circuit DAC is needed.

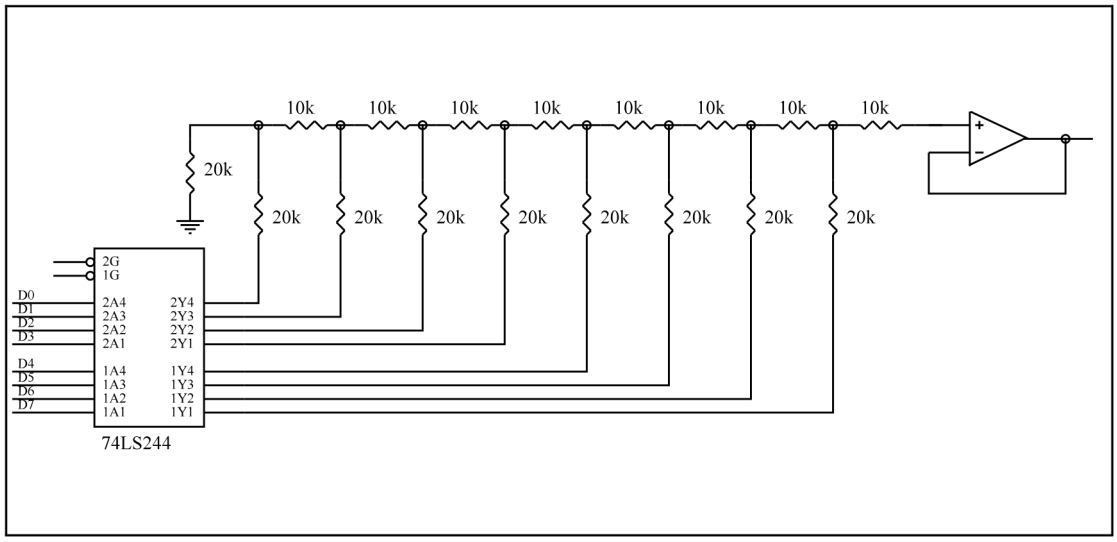


Fig 7.2 - Implementing a Resistance-ladder network

A more expensive option is to use a readily available converter such as the MC1408, shown in Figure 7.3. Note that the chip only takes the place of the ladder network and data switches. You still have to supply the reference voltage, amplifier, and other discrete components. But other DACs will have a data latch built in as well.

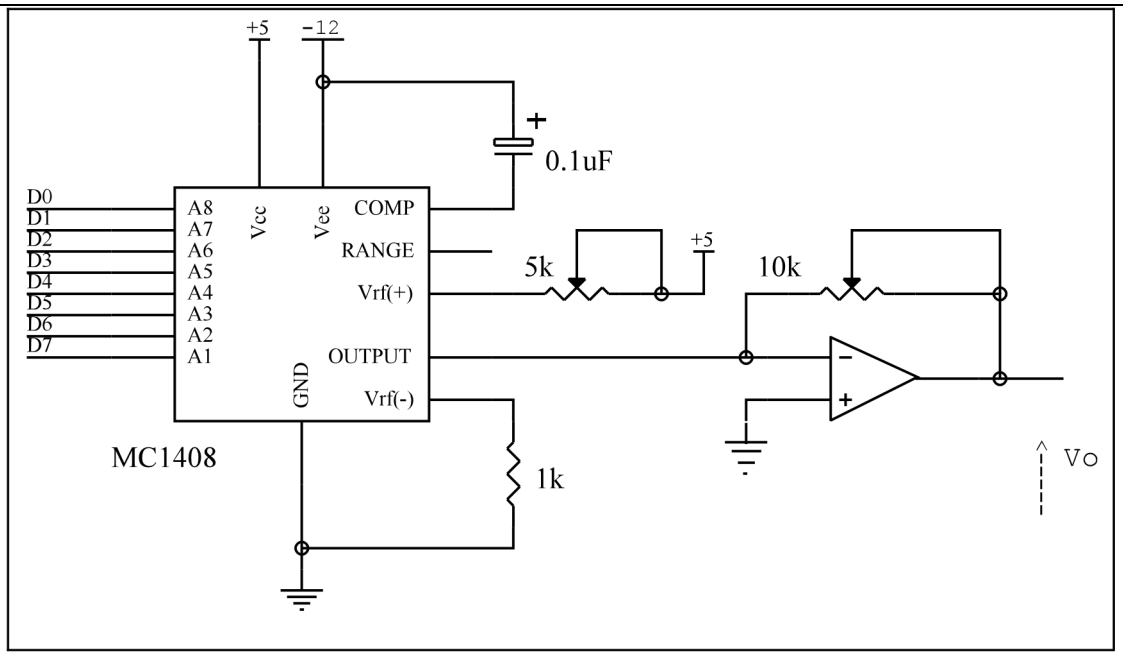


Fig 7.3 Interfacing a MC1408 DAC

Due to the discrete nature of the digital to analog process, it should be noted that the analogue output waveform is a stepped. For example, a sawtooth waveform comes out as a series of steps. Sometimes a low pass filter, also known as a reconstruction filter is used to smooth out the waveform.

7.2.1 Generating waveforms

In many applications we have to create an analogue periodic waveform digitally. This process is called synthesis. We then manually compute the digital values and put these into a lookup table, or even put it into an EPROM. There are three factors to consider concerning the waveform:

a) The maximum analogue value
b) The output frequency
c) The number of samples per period

The first factor concerns the quantisation error which determines how much the digital value approximates the analogue equivalent. We want to use the full range of the analogue signal to correspond to the full range of allowable digital values. This minises the quantisation error which determines the scaling factor, which we will discuss shortly. Also, we must know if the DAC is capable of handling negative voltages. If not, we may have to add in an offset voltage so the minimum value of the waveform is zero. For example, sinusoidal waveforms need this offset.

The second factor is limited by the execution speed of the processor. The third factor is a combination of the two.

7.2.1.1 Generating a Sine Wave

In theory the DAC accepts inputs from 0 to 255 and outputs proportionally from 0 to the maximum analogue voltage. But because of the components used, there may be clipping after a certain voltage.

Assume we have an 8-bit, DAC in which the supply voltage is 5V. We expect the DAC to produce analogue outputs from 0 to 5V. We want to output a sine wave with a 4V peak to peak voltage. Digitizing with 8 bits does not allow negative values. So we need to offset the sine wave. In this case, we will add an offset so that the minimum voltage is 0V. By doing so, we make full use of the voltage range that can be digitized. This will reduce quantization noise.

For a sine wave, the amplitude is half the peak to peak voltage or 2V. The offset is also 2V. So the equation of the sine wave is:

$$V = 2(1+ \sin \theta)\ V$$

where θ depends on the number of samples per cycle. More generally, $V= V_{amp}*\sin \theta + V_{offset}$ as shown below.

NOTE: **This discussion is true for SINE type waves only!**
Of course, digitizing other waveform shapes require knowledge of the equation of the waveform.

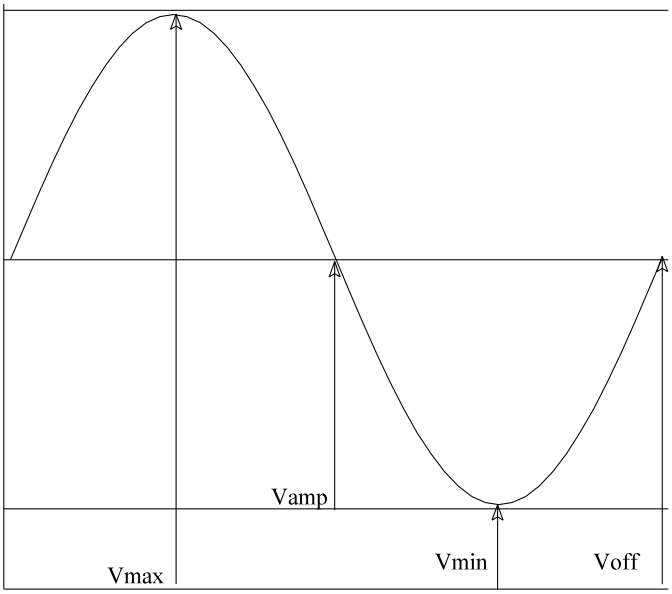


Fig 7.4 Sine Wave

Whenever possible, we should let the lowest value of the waveform be equal to zero, for ease of computation. In some cases this is already true. For example a rectified sine wave does not have an offset, as its lowest value is zero.

Resolution

From the description of the analogue wave, find out its full scale value. This should be equivalent to the largest digital value the D/A converter can handle. In this case, since we have an 8 bit DAC which outputs from 0 to 5V, an input of FFH (255_{10}) will result in 5 volts being output. Thus for 1 volt, the equivalent digital value is:

$255/5 = 51_{10}$. We call this the scale factor.

Samples per period

Now let us specify that we want 24 samples per period. We need to compute the value of the sinusoid at each sample point. Note that one period of a sinusoidal wave is equivalent to 360 degrees. So, to compute the value of the sinusoid at each sample point, we will divide 360 by the number of samples. A full treatment of the samples per period will depend things like the Nyquist criterion, which states that the sampling frequency should be twice that of the highest frequency present in the signal.

Here, we will focus on the shape of the waveform and ease of calculation. Less samples per cycle gives a more digital output as mentioned above. In this case let us consider 24 samples as giving us an acceptable likeness to a sine wave. So in one cycle of the sine wave, we need to calculate the values of the sinusoid at 360/24 = 15 degree intervals. It is convenient to do this in the form of a table, as shown below:

S/no	2	sin 2	V = 2(1 + sin 2)	$F_{scale} * V_{10}$	Rounded
0	0	0	2	102	102
1	15	0.2588	2.5176	128.3976	128
2	30	0.5	3	153	153
3	45	0.7071	3.4142	174.1242	174
4	60	0.866	3.732	190.332	190
5	75	0.9659	3.9318	200.5218	201
6	90	1	4	204	204
7	105	0.9659	3.9318	200.5218	201
8	120	0.866	3.732	190.332	190
9	135	0.7071	3.4142	174.1242	174
10	150	0.5	3	153	153
11	165	0.2588	2.5176	128.3976	128
12	180	0	2	102	102
13	195	-0.2588	1.4824	75.6024	76
14	210	-0.5	1	51	51
15	225	-0.7071	0.5858	29.8758	30
16	240	-0.866	0.268	13.668	14

17	255	-0.9659	0.0682	3.4782	3
18	270	-1	0	0	0
19	285	-0.9659	0.0682	3.4782	3
20	300	-0.866	0.268	13.668	14
21	315	-0.7071	0.5858	29.8758	30
22	330	-0.5	1	51	51
23	345	-0.2588	1.4824	75.6024	76

Output Frequency

This is equal to 1 / (waveform period). The waveform period is the number of samples per period *multiplied* by the time between samples. In the above example, the number of samples per period is 24. If the time delay between each sample is 50 :s for example, the frequency of the sine wave above will be 1 / (24 * 50 :s). If we have a large number of samples per period, we will need a shorter time period between samples in order to get the same frequency. This time period between samples has a minimum value which is determined by the speed of the processor.

The question now is how to send these numbers to the DAC in order for it to produce the waveform. This can be done by either software or hardware number generation.

7.2.1.2 Software Generation

One option would be to write a program with the numbers stored as a look up table. Each data would be fetched in turn and sent to a port with the DAC attached to it. The numbers would create the basic profile of a Sine wave, and the total period of the wave would be determined by the time period between fetching and sending each individual number. The maximum frequency of the Sine wave would be limited by the speed of your software. It might be possible to generate the numbers using a software algorithm, however, this would severely limit the rate at which the numbers could be produced, though it might save memory space if the look up table was going to be lengthy and yet the algorithm is simple.

Here is how a C language program would output these values:

```
#define     Count       23    /* number of values in table */
#define     DACPort  *((char *) 0x133          /* DAC Port  */

const unsigned char WaveTable[Count] =
{    102 102,128,153,174,190,201,204,201,190,174,153,128,
      102,76,51,30,14,0,3,14,30,51,76
};

void main (void)
{                          /* main entry for program */
      unsigned char i;
      while 1
```

```
        {
              for (i = 0 ; i <= Count; i++)
              {
                    Sleep(1);        /*delay */
                    DACPort = WaveTblPtr[i]; /* op to data port */
              }
        }
}
```

7.2.1.3 Hardware Generation

A ROM based waveform generator is an alternative way of sending the numbers to the DAC. You would program your look up table into an EPROM and use a counter to create the addresses. In the following example, although we have an 8k device, we will only be using the first 256 locations as we only have an 8 bit counter. The content of address 00H would be 102 decimal, address 01H would be 128 decimal and so on. Thus we will use up to 256 samples in the waveform.

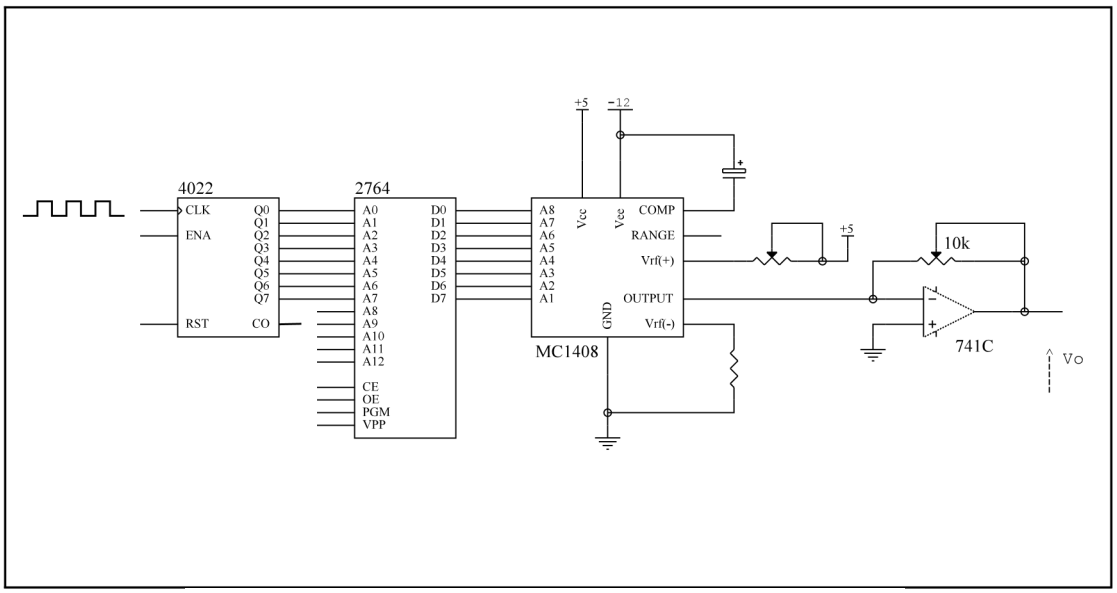


Fig 7.5 A ROM based waveform generator

7.2.1.4 Pulse Width Modulation

The previous methods discussed need a latched output with many pins and also analogue resistors which need high precision. Respectively these result in added complexity and manufacturing cost.

For a given sampling period, pulse width modulation (PWM) uses digital voltages and represents an analogue value by the proportion of *time* the voltage is on. In this case, one digital output can produce one analogue voltage. In the figure below, the left side shows a voltage V_{avg} being generated by a fixed digital voltage V_{hi}. Assuming V_{hi} is 5 volts and if the duty cycle is 80%, by filtering the waveform, we can get V_{avg} being 4 volts. On figure 7.6, we see the PWM waveform increasing in duty cycle as the analogue waveform ramps up.

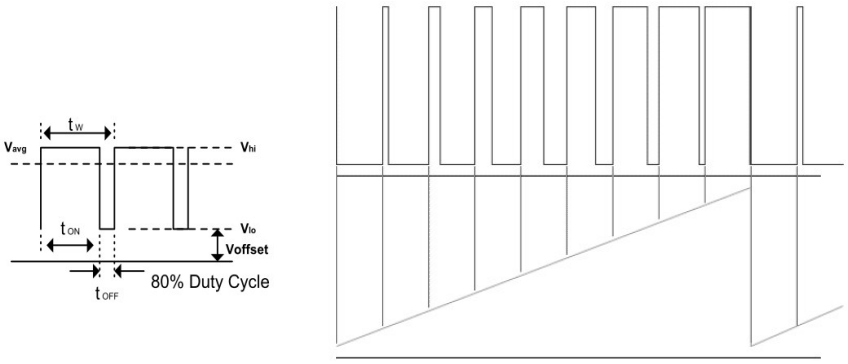


Fig 7.6 Converting a Pulse-width modulation digital output to a DAC output

The benefits are:
- lesser pin count which translates to a smaller size
- lower heat dissipation as the PWM outputs are either On or Off,
- lower cost as there is no need for expensive analogue processes
- better noise immunity because of digital signals
- high switching speeds

However, there is the problem of ripple and digital noise caused by the switching and the need for good filtering techniques.

7.3 Analog-to-Digital Conversion Techniques

Analog-to-Digital Converters (ADC) are systems that are used to convert analog signals into digital equivalents. The main techniques are:

- Integrating
- Counter Ramp
- Successive Approximation
- Flash
- Sigma-Delta

7.3.1 Integrating Techniques of AD conversion

The principle used by the intergrating techniques is to let an integrator be charged towards a required voltage. Generally, the larger the input voltage the longer the charge time. The output is controls a counter which provides a digital representation of the time required for the conversion, which is directly proportional to the magnitude of the input voltage.

Consider an analog process where switch S2 is first closed to discharge the capacitor C. When the capacitor is discharged, S2 is opened and S1 is closed, at the same time, the counter is reset. The capacitor then charges up via $-V_{ref}$ and a ramp voltage forms at the input of the comparator. This ramp is compared to the input voltage V_{in}. When the ramp voltage is larger than V_{in}, the comparator switches off the counter. The number of counts is proportional to the magnitude of V_{in}.

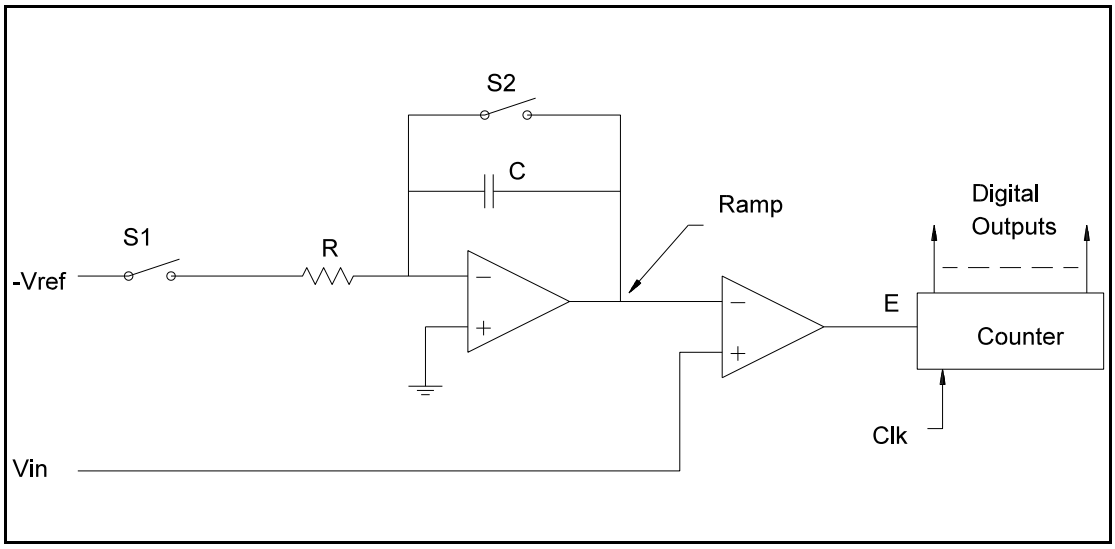


Fig 7.7 Analog integrator

The single ramp ADC has some limitations as its accuracy depends on:

a. the resistor, R which can drift over time and temperature
b. the capacitor, C which can also drift over time and temperature
c. the reference voltage, $-V_{ref}$
d. the device timing process

These problems can be overcome by using a Dual Slope ADC which will not be covered.

7.3.2 Counter Ramp ADCs

The simplest in concept is the Counter Ramp ADC. The digital logic circuit is a counter and a comparator. Figure 10-9 shows the block diagram of the Counter Ramp ADC.

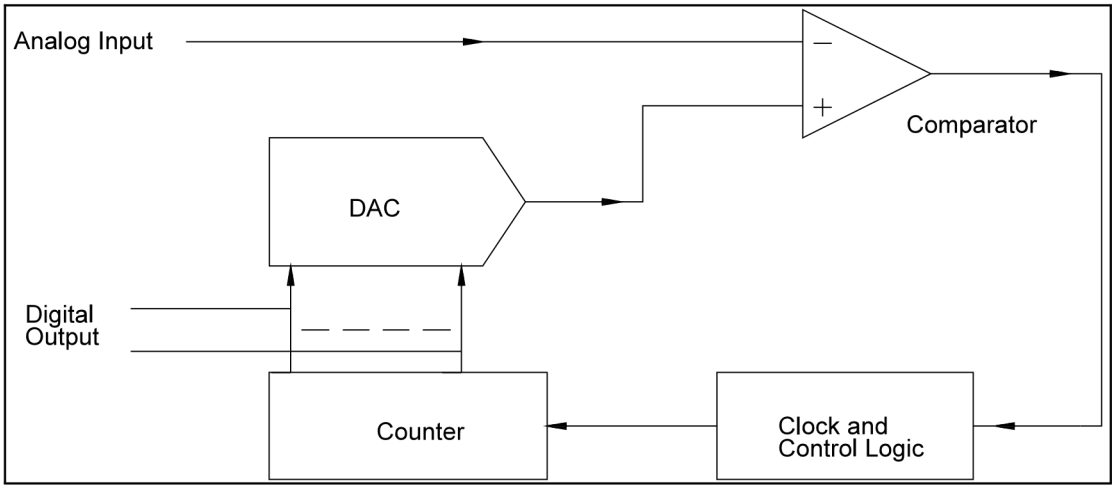


Fig 7.8 A counter ramp ADC

At the start of a conversion, the counter is set to zero and hence forces the DAC output to 0 volts. The non-inverting input of the comparator is thus 0 volts. If the input analog signal at the inverting input is non-zero, the comparator output is set to a logic '0' which enables the clock to pass to the counter. The counter increments in steps, thereby increasing the values output from the DAC.

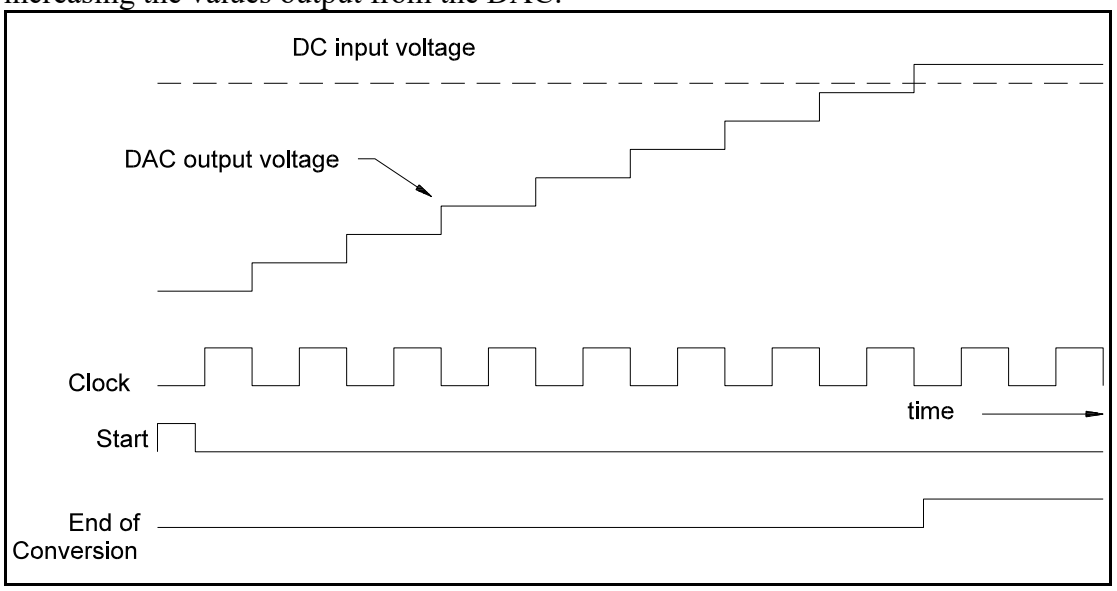


Fig 7.9 Timing Sequence for Counter Ramp ADC

This process continues until the clock is disabled, ie, when the output of the DAC is larger or equal to the analog input. Figure 10-10 shows the timing sequence of the operation.

Note that the counter increments on the rising on the rising clock edge and that the comparator output changes when the DAC output exceeds the input voltage. At the end of the conversion, the comparator outputs a logic "1" which is detected by the control logic. The control logic then generates an **End-of-Conversion** signal to indicate to the digital subsystem that the binary representation of the output can be read from the counter inputs.

A disadvantage of the Counter Ramp is the long conversion time required to count up from the value of 0 volts to the input value. A faster clock may be used to speed up the process but consideration has to be taken regarding the settling time of the clock signal as well as the response of the overall circuitry. Furthermore, smaller input voltages will be converted faster than larger ones. This leads to different conversion times for different voltages.

7.3.3 Successive Approximation ADC

The Successive Approximation ADC provides a more rapid conversion than the Counter Ramp ADC. Rather than increasing the DAC output one bit at a time, the digital logic circuitry uses a binary search. This works by taking the value to be found and comparing it with half of the maximum range of possible values. Depending on whether it is larger or smaller than this value, it is accumulated, and another comparison will be done on the lower or higher half of the half range. This is done until only one bit is left to be compared. The number of comparisons is the same as the number of bits.

The following figure shows the block diagram of an eight-bit Successive Approximation ADC. The major difference from the Counter Ramp ADC is that the counter has been replaced by an eight-bit register. The states of the output bits of this register are controlled by the clock and control logic block.

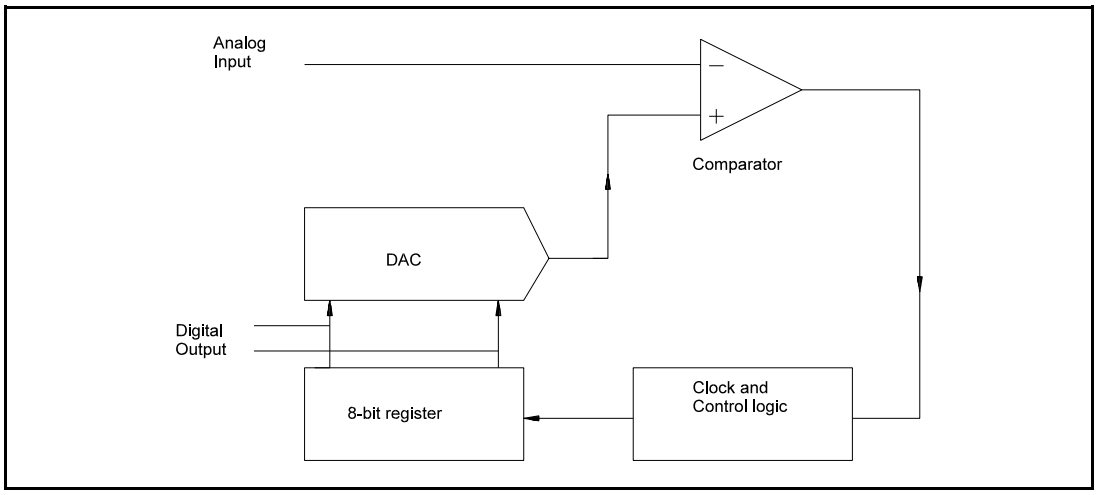


Fig 7.10 A Successive Approximation ADC

As an example, consider a DAC which gives 5 volts is when we input FFH to it. Assume an analogue voltage of 3.57 volts is applied. The value the ADC should give for this voltage is: (3.57/5) *255 = 182_{10} or B6H. Let us see how this is value is obtained.

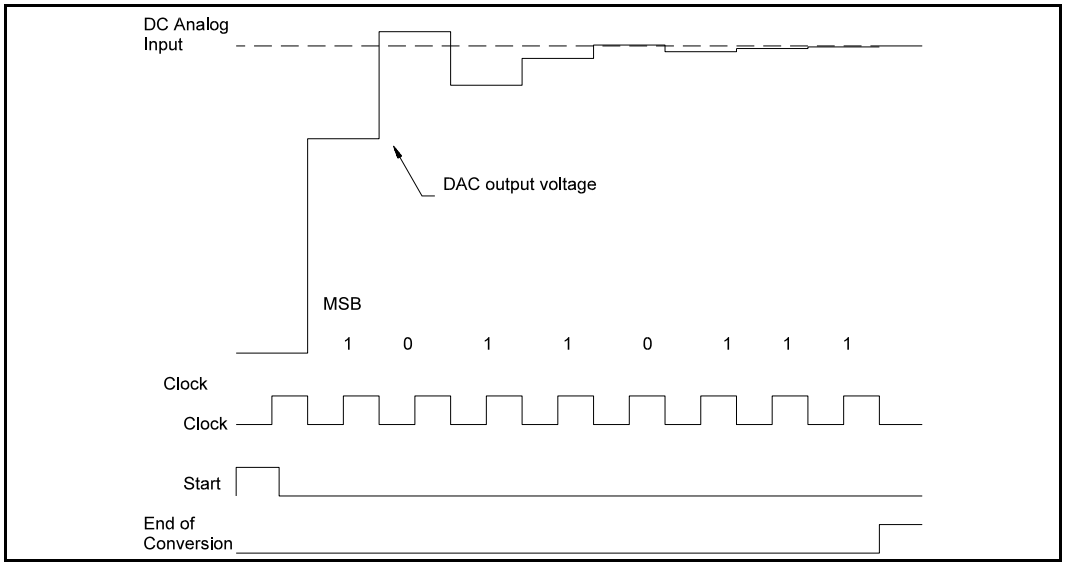


Fig 7.11 Timing Diagram for the Successive Approximation ADC

Register value-bit ptr	Decimal	DAC voltage	< Input voltage?	Action
*1*000 0000 - bit 8	128	2.51	Yes	Keep bit
1*1*00 0000 - bit 7	192	3.76	No	Clear bit

10*I*0 0000 - bit 6	160	3.13	Yes	Keep bit
101*I* 0000 - bit 5	176	3.45	Yes	Keep bit
1011 *I*000 - bit 4	184	3.61	No	Clear bit
1011 0*I*00 - bit 3	180	3.53	Yes	Keep bit
1011 01*I*0 - bit 2	182	3.57	Yes	Keep bit
1011 011*I* - bit 1	183	3.59	No	Clear bit

So the digital value is 182_{10} or B6H. In summary, the algorithm is as follows:

1. Clear all bits to zero, place bit pointer to Most Significant Bit
2. Set bit indicated by pointer
3. Use DAC, generate analogue voltage
4. Compare with Input Voltage
5. If Input Voltage > Generated Voltage goto step 7
6. else clear bit indicated by pointer
7. Bit pointer positioned to next lower bit
8. If all bits in register processed then stop, else go to step 2

Of course, we can implement the digital logic and clock as a piece of software from a processor. The comparator and DAC is still needed however.

Parallel/Flash

ADCsThe **Parallel** or **Flash** ADC is the fastest ADC currently available, they use a resistive potential divider to generate the series of voltage levels. There are 2^n-1 levels for a N-bit conversion.

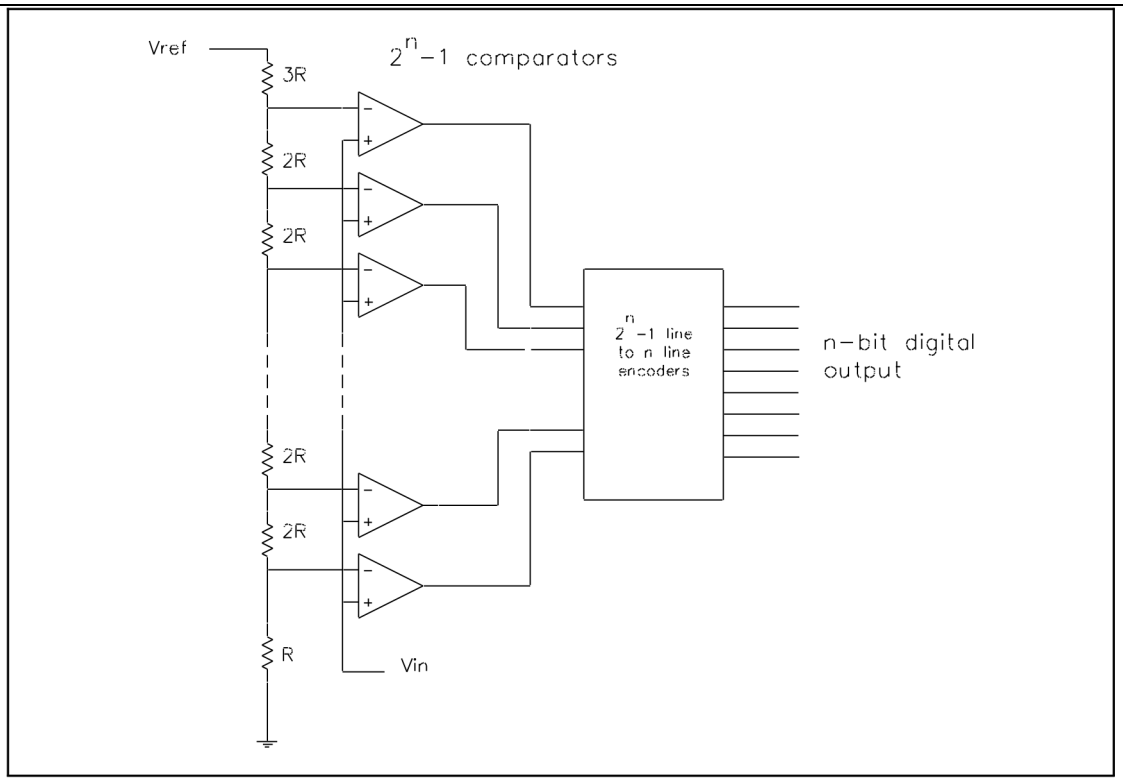


Fig 7.12 A parallel ADC

The parallel conversion principle is conceptually straight forward, but it has the disadvantage of requiring a large amount of circuitry to implement a high resolution conversion. An 8-bit system would require 255 comparators!

7.3.4 Sigma Delta converter

This method of conversion is relatively recent, following developments in Integrated Circuit technology which allow for greater amounts of digital circuitry to be included for better signal conversion characteristics. The method is a development of PWM in using one bit for sampling purposes, but at a frequency much higher than that of the input. Instead of turning signals on and off for fixed periods, we can generate a ramp by summing a series of 1's and 0's and approximate any value we want, at a fast enough rate. This is called oversampling, which "spreads out" the quantisation noise (see later) over the higher sampling rate.

Consider a 3 bit representation of a signal which uses $2^3 = 8$ levels over 8 time samples. As before, if we use a Counter Ramp type of conversion (see middle waveform) we can approximate the analogue signal with a series of Pulse Width Modulated (PWM) signals.

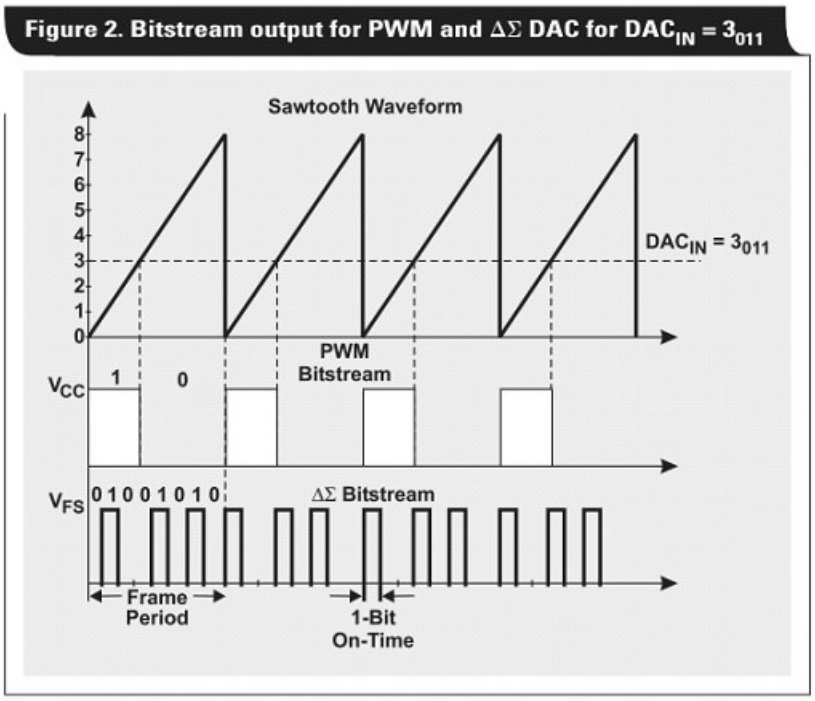


Fig 7.13 Comparison of PWM with Ó-Ä data conversion

In contrast, if now we use a series of pulses (lowest waveform) at a higher frequency to represent the signal over the *same* period of time, we can achieve the same amount of energy transmitted- but we need to *average* over that time period. Consider that if we now have *more* samples in the same time interval, we can more accurately approximate the analogue signal. This requires hardware that can work at a high frequencies but the benefits are that:

- it easier to filter quantisation noise as it occurs at a high frequency
- allows for more pulses to represent a signal by *averaging* it over a higher sampling rate thus achieving higher resolution without the need for high precision analog circuitry - moving more of the quantisation noise to higher frequencies is also known as noise shaping

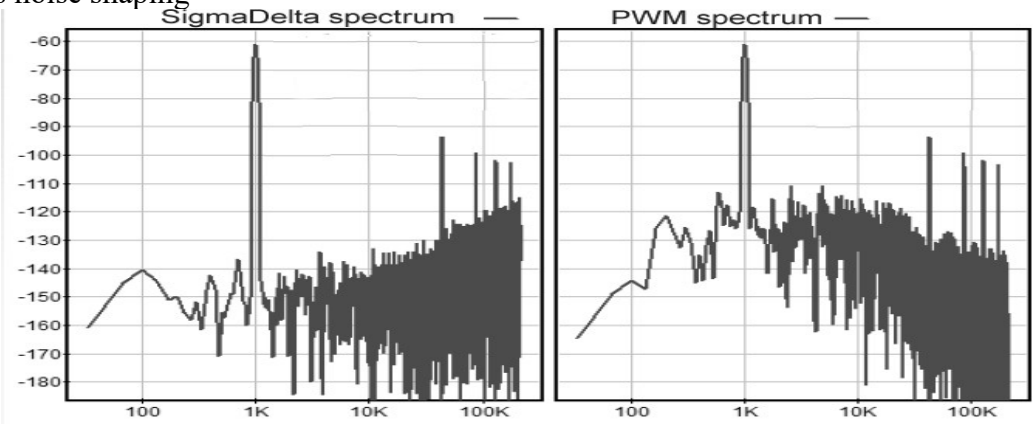


Fig 7.14 Comparison - PWM (left) / Sigma-Delta (right) spectrum of 1 kHz sine wave

From the figure, we see that at the -150 dB point, the Sigma-Delta converter has unwanted signals till about 10kHz when the amplitude of these signals rises, making it harder to low pass filter away. But the PWM has these spurious frequencies at 1-20 kHz

at a higher amplitudes, making it hard to filter away. Together with the noise reduction a higher resolution is achieved, easily giving 16 bits and often 24 bits.

The block diagram shows a first-order Sigma-Delta ADC device, the details of its operation are in the appendix.

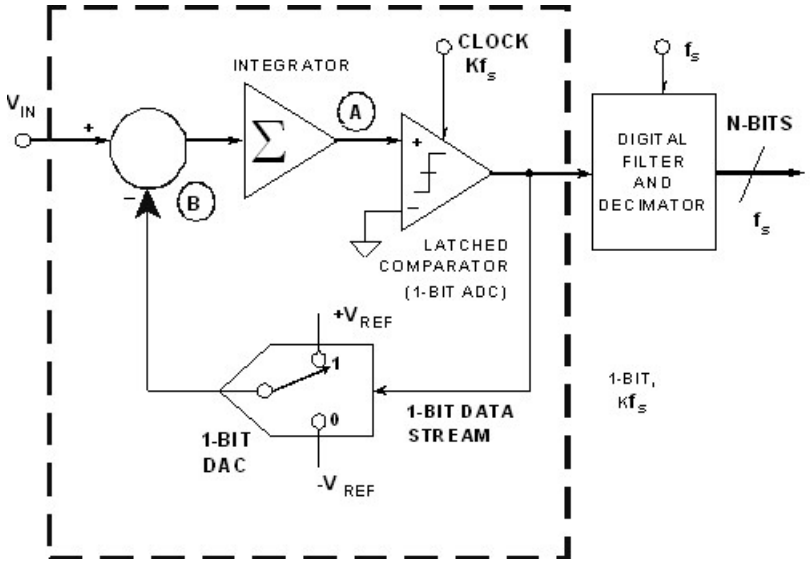


Fig 7.15 Block diagram of a first order Sigma-Delta ADC

Sample Hold Amplifier

A sample/hold amplifier is a discrete or integrated device placed in between the input waveform and the ADC. Its purpose is to hold Vin as a constant whilst the conversion is in progress, so that the ADC is not following a changing waveform. Conceptually it is an electronic switch and a capacitor. If no conversion is required the switch is closed and the voltage across the capacitor follows Vin, when conversion starts the switch opens and the voltage across the capacitor will remain constant until conversion is complete.

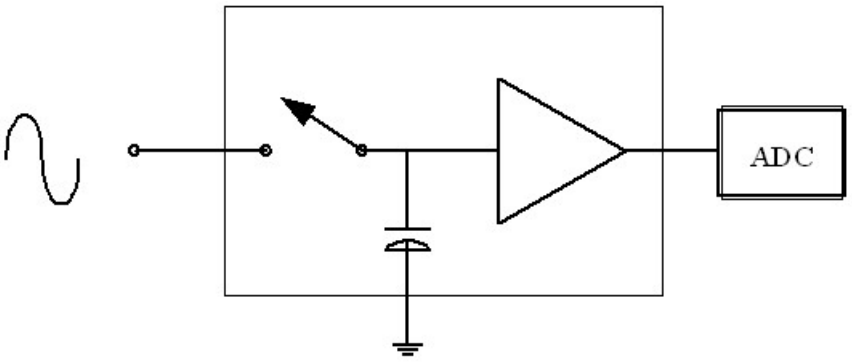


Fig7.16 The inclusion of a Sample/Hold Amplifier

7.4 Errors in DA and AD Conversion

There are several sources of errors when converting data. Some are due to the very nature of the process, others are due to impefections in the electronic components.

7.4.1 Quantization Errors in D/A Conversion

The analog output of a DAC is not infinitely variable, but is quantized into small but definable steps. Hence the voltage produced at the output of a DAC is seldom able to be exactly the same as the voltage at the output of a true analog system.

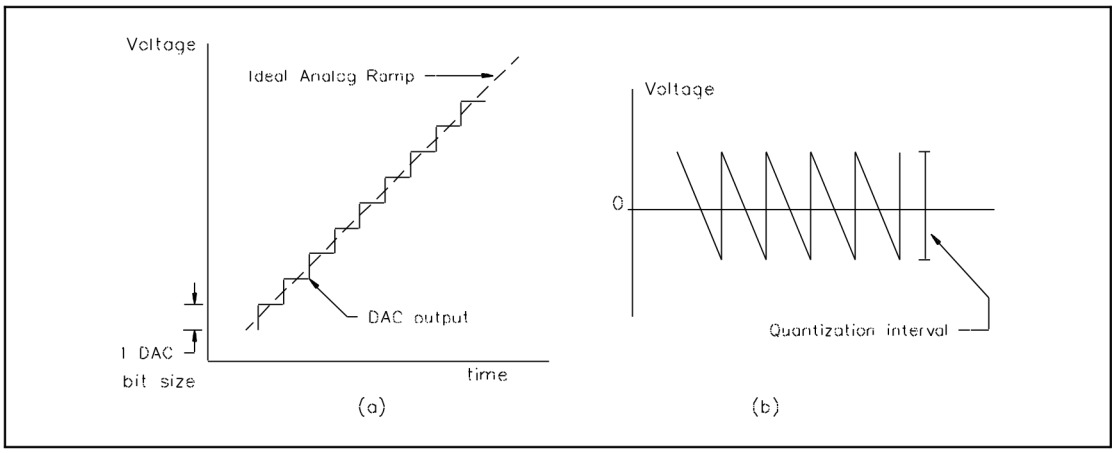


Fig 7.17 Quantization Errors

A portion of the DAC output waveform is superimposed onto an ideal analog waveform. The difference between these two voltages represent an error in the system. As this difference in voltage arises from the quantized nature of the output of the DAC, it is called **Quantization Error**.

To reduce the quantization error, a DAC must be controlled by more input bits. If the full-scale voltage remains the same, the increase in bits will reduce the quantization interval. Quantization error is also often referred to as **Resolution**.

Resolution can be expressed in several ways.... eg for an 8-bit A/D:

- as a number of bits ..... 8-bit resolution.
- as a percentage ..... 0.39% ($1/2^8 * 100$)
- as the relative size of the LSB..... 1 in 256
- as an absolute value (say 5v range)... 19mV ($5V/2^8$)

APPENDIX

Other Sources of Errors

Gain Error

This is often caused by errors in the feedback resistor on the converter op-amp. The error is usually a linear scale to the expected output.

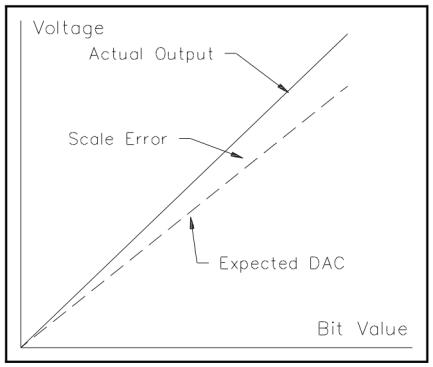


Fig 7.18 Gain Error

Offset Error

This is when the output is non-zero when a zero input is used. This error can be caused by op-amp errors or leakage currents in the current switches. The error can be corrected by taking readings at zero input value and subtracting from the output.

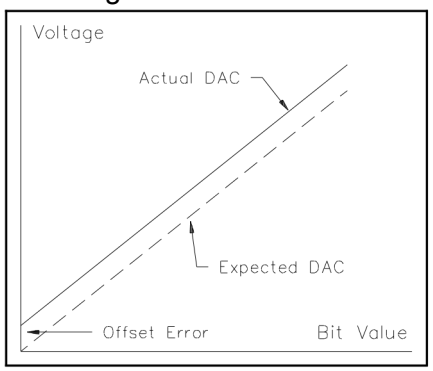


Fig 7.19 Offset Error

Linearity Errors

This error is usually caused by errors in the current source resistor values. Hence the output loses its proportionality.

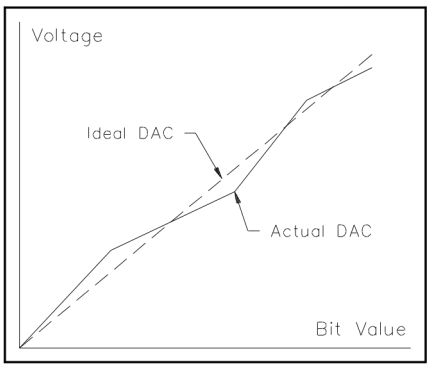


Fig 7.20 Linearity Error

Internal detail of D/A converters

The structure of the network is balanced so that the current supplied from any switch is halved at every junction. A binary relationship between each of the switches and hence the current resulting from the switches can be used to represent binary numbers.

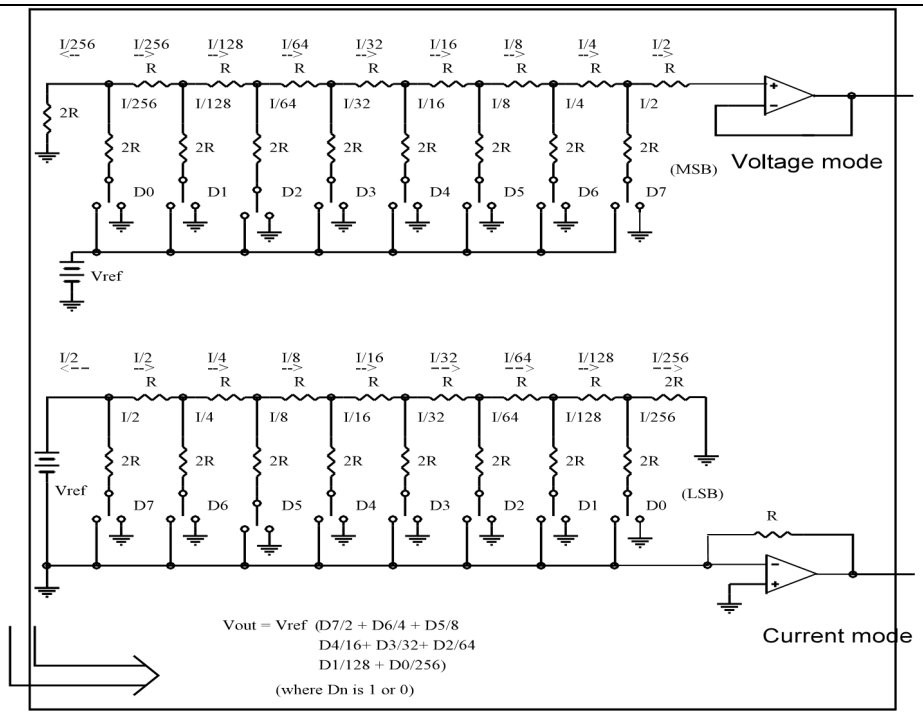


Fig 7.21 A Theoretical Resistance-ladder network

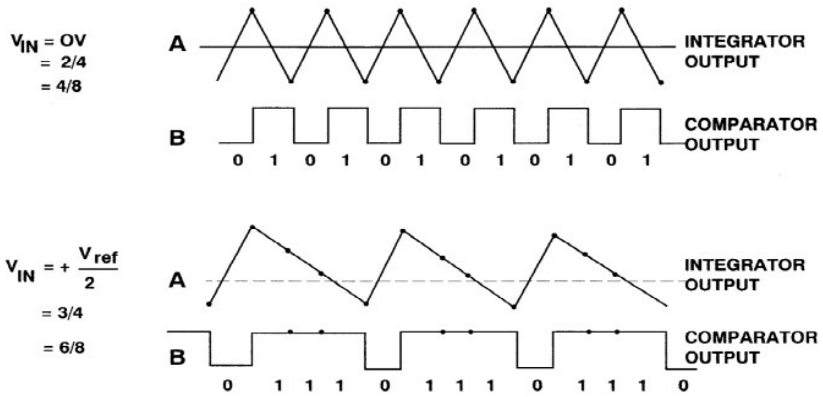
The above figure shows two practical and low cost methods of implementing a DAC which is widely used. The R-2R network is implemented with discrete resistors. A buffer or latch is used to drive the resistor network. An opamp which is capable of 0 volt output is used. This circuit may be used for many applications without much loss in quality. As shown, we can use the network in two modes. The voltage mode is easier to implement and understand. Current mode produces less noise and does not need a high performance op-amp buffer and is preferred in integrated circuit versions of the R-2R ladder.

Details of ∑-Δ (Sigma-Delta) conversion

Assume a DC input at V_{IN}. The integrator will be always incrementing or decrementing at point A. The output of the comparator is fed back through a 1-bit DAC to the summing input at point B. This negative feedback will keep the average DC voltage at point B be equal to V_{IN} which is done by controlling the average DAC output. The average DAC output voltage is controlled by the amount of 1's in the 1-bit data stream from the comparator output. As the input signal increases towards $+V_{REF}$, the number of 1's in the serial bit stream increases, and the number of 0's decreases and vice versa. In this very simple description, this analysis shows that the average value of the input voltage is can be represented by the bit stream out of the comparator. The digital filter and decimator process the serial bit stream and produce the final output data.

Unlike the other types of ADCs considered, only when a large number of 1-bit ADC samples are averaged, will a meaningful value result. The ∑-Δ converter is difficult to analyze in the time domain because of this apparent randomness of the single-bit data output. If the input signal is near positive full-scale, it is clear that there will be more "1"s than "0"s in the bit stream. Likewise, for signals near negative full-scale, there will be more "0"s than "1"s in the bit stream. For signals near midscale, there will be approximately an equal number of "1"s and "0"s. Figure 5 shows the

output of the integrator for two input conditions. The first is for an input of zero (midscale). To decode the output, pass the output samples through a simple digital lowpass filter that averages every four samples. The output of the filter is 2/4. This value represents bipolar zero. If more samples are averaged, more dynamic range is achieved. For example, averaging 4 samples gives 2 bits of resolution, while averaging 8 samples yields 4/8, or 3 bits of resolution. In the bottom waveform of Figure 5, the average obtained for 4 samples is 3/4, and the average for 8 samples is 6/8.



(Vin-DAC) Adder	Integrator out (V)	Comparator (DAC in)	DAC out (V)
1 (Vin=1 or $V_{ref}/2$)	0		0
1	1 (0+1)	1 (1>=0)	2

-1 (1-+2)	0 (1+-1)	1 (0>=0)	2
-1(1-+2)	-1 (0+-1)	0 (-1<0)	-2
3(1- -2)	2 (-1+3)	1 (2>=0)	2
-1(1-+2)	1 (2+-1)	1 (1>=0)	2
-1(1-+2)	0 (1+-1)	1 (1>=0)	2
-1(1-+2)	-1 (0+-1)	0 (-1<0)	-2

8 SYSTEM DESIGN AND UML

8.1 Introduction

Microprocessors can be programmed to perform a wide range of tasks. In fact, they must be programmed before they can do anything at all! As they are fabricated using Large Scale Integration (LSI) technology, the unit cost of each processor is very low. However, the development of systems using microprocessor tends to be high. Proper design techniques would help reduce this developmental cost. That is the aim of this chapter.

8.1.1 Why microprocessor-based?

The microprocessor offers opportunities for products and product features which are not achievable by other means. It is difficult to achieve the range of features found in today's electronic devices. For example it would be almost impossible and not economical to design a Compact Disk player using logic gates alone.

8.1.2 Implications of Incorporating a Microprocessor

Testing the microprocessor based system is a huge hurdle one needs to overcome. Firstly, software needs to be extensively and thoroughly tested. If the underlying fault arises because of the operating system, changing it is often impossible. This is one of the problems that could arise from software testing.

Secondly, the hardware of the system is almost as difficult to test as the software. This is because hardware interacts with software and it might be difficult to isolate whether an error arises from hardware instead of software.

As microprocessors only deal with digital data, an interface must exist between the microprocessor and its peripheral devices to convert data, should the format be different.

8.2 Product Development

Trends in electronics industry

Even though products tend to have a shorter product life, every new version is expected to have a higher performance over price ratio due to the intense competition in the market.

As a result, manufacturers need to

1. shorten product design cycle using improved design methodologies and

2. invest on tools & resources to improve and increase product features.

8.3 Product Design and Development Activities

Product design is a human process. Design is a process which involves communication, creativity, negotiation and agreement. The following diagram gives the approach we will take in this chapter.

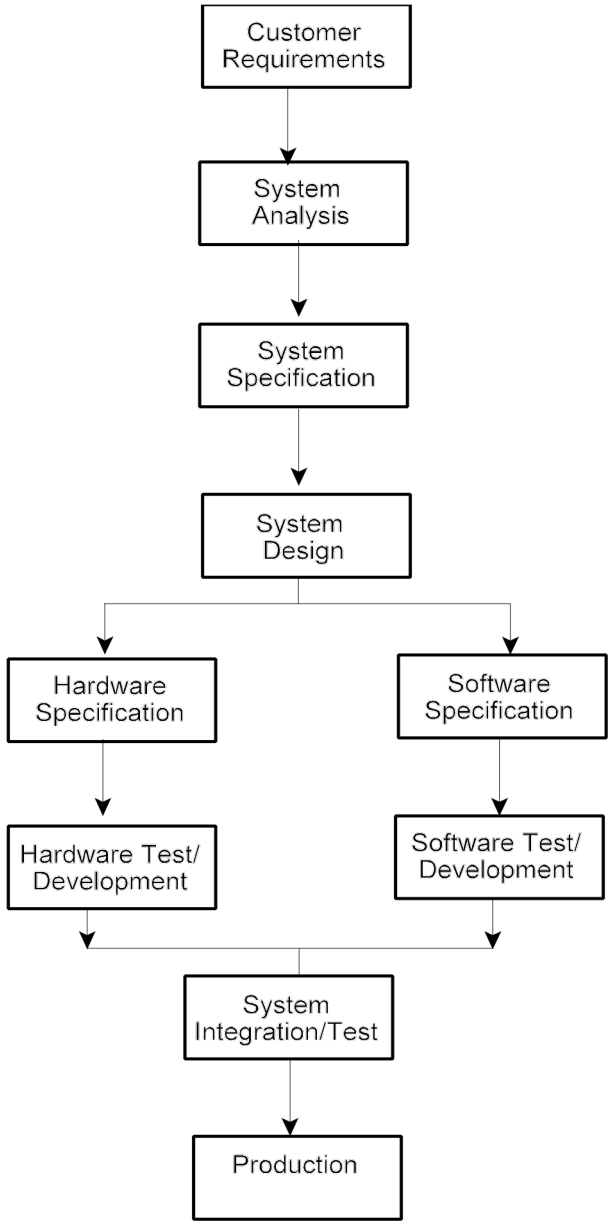


Fig 8.1 Product Design & Development Activities

System designers do not work in isolation. There is a strong temptation to sit down and produce the system at the earliest possible moment. This is the cause of much of the problems in the software engineering industry. The hardware designers and the software

designers need to understand each other so that the final product meets requirements of the client. Very often in the design and production process, the communication, negotiation and agreement aspects of a design process are left out.
A notation which is clear, consistent, and which can be used to communicate within a system development team, and with the clients and other third-parties which the team needs to deal with, will certainly come in useful.

Systems were constructed using a "waterfall" approach traditionally. As water flows down from the top, this approach starts off by having clients formally agree on a requirements document. The designers would then come up with a design which would be further agreed by the clients. The system would finally be implemented, and what follows would generally be an endless process of maintenance.

Modern ideas move away from this. Many system developments use instead the iterative method. The method consists of developmental cycles with each cycle making up of analysis, design and implementation. Each subsequent cycle build on earlier cycles. One such method is the Unified Modeling Language (UML). It is a language for specifying, visualizing, constructing, and documenting the features of systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.

The UML represents an important part of system development process. UML uses mostly graphical notations to express the design of system projects. Using UML helps project teams communicate, explore potential designs, and validate the architectural design of systems and thus we will use it in our product design.

8.4 Product Requirement

The first stage in the both the iterative method and the "waterfall" method is requirements gathering. The difference between the two is that the iterative method is only interested in capturing the most important features in the beginning and allowing subsequent cycles to handle the less important ones; the waterfall method however starts off with the intention of obtaining all the features of the system. In any case, this initial description of the product's intended functions is obtained from the customer. One is only interested in how the system will interact with its intended user and its environment at this stage.

8.4.1 Need for Product Requirement

There are two reasons why one needs the requirement gathering phase:

Avoid 'creeping featurism'

Because of the choices available to the designer when developing microprocessor based products, there is always the desire to interfere continually with product specification during development. This can significantly alter the cost and time required for product development

Avoid 'missing features'

The worst that could happen are that some required features were missed out, and when noticed, it is either too late or a major redesign is required.

8.4.2 Goals and Constraints

These are additional information from the customer apart from product requirement description.

Goals

These are aspects of customer requirements which guide the system designer where freedom of choice exists. For example, Hardware cost of the motor interface be minimized

Constraints

Aspects of customer requirement which limit the freedom of choice of the system designer. For example, The product should include a particular type of component.

Constraints narrow the scope of the project. Goals are customer preferences to be considered so that the product can function withing the given constraints.

- Goals and constraints may affect the same design feature
- There should not be any constraints that restrict the freedom of choice of the designer unnecessarily
- Avoid contradiction in the list of goals

Types of Constraints

AREA OF CONSTRAINTS	EXAMPLES
Microprocessor	Particular type specified by customer
Other hardware	Some or all of additional hardware specified
Field-repairable	PCBs must be plug-in
Power Supply	Must run from batteries
Environment	Must be insensitive to radio interference

Size	Must fit into particular box
Unit Component Cost	μP and interfaces must cost less than a particular limit
Development Cost	Amount available for supporting equipment, consultants, etc
Development Time	Prototype available for demo at an exhibition

So in the above example, a constraint would be the use of a waterproof chassis.

8.5 System Analysis

System analysis consists of activities that include examination by the system designer of the initial customer requirements. Some considerations when performing system analysis are:

Subsystem Identification
From the requirements, we try to identify hardware components or sub-systems in the product.

Dynamic modeling
Dynamic modeling tries to capture how the parts of the system behave and how they interact between each other.

Feasibility studies and/or simulation
If system designer simply does not know whether the product can be designed to specifications, the use of feasibility studies and/or simulation can remove much of the uncertainty without the expense of designing to the prototype stage.

8.5.1 Sub-system Identification

In this stage, we need to identify the components that would make up the whole system. These are components that perform a function and does not have to be extensively programmed by the designer. The advantage of subsystems are that they are complete products whose behaviour and cost are known. Typical examples of sub-systems in embedded systems are the microprocessor, LCD, keypads and stepper motors.

8.5.2 Dynamic Modeling

A task is a part of the overall product function which is to be provided by a suitably programmed microprocessor system. We need to obtain from the customer requirement a statement of what the system has to do.

Then we use dynamic models to analyse the behaviour of systems. Dynamic modeling is similar to a form of story telling. The approach we are going to adopt is to look at the functions of the system we are analyzing, thinking through them in terms of scenarios and then see how these scenarios affect each of the individual sub-systems.
The output of this phase is the production of interaction and use case diagrams. Detailed examples are given later.

8.5.3 Feasibility Studies/Simulation

At the analysis stage, it is important to have an idea if a particular algorithm has a chance of working. This should be done without too much expense or hardware construction.

8.5.3.2 Feasibility studies

These are special-purpose activities, aimed to answer a particular question or group of questions about an aspect of the customer requirement.

Some questions to ask are: can it meet project constraints, is the performance of the hardware system sufficient, or does the algorithm work correctly.

There may be some hardware or software design, but should be easily done, as compared to building a full prototype of the proposed product. The prototype is designed to answer most of the remaining questions about the product.

8.5.3.4 Simulation

These are means of exploring, rapidly and easily, various alternative answers to more general questions. Very often they are done in software.

- Simulation is used to build system which in some ways, behaves like the actual product.
- Various versions of the proposed system can tried out without extensive redesign.
- Principal use of simulation in µp-based product development is for evaluating competing algorithms. If these are complex, we may not be able to evaluate them based on logic or on manual calculations.

8.6 System Specification

This is produced as a result of system analysis, by the designer. It is the overall controlling document in a project. The UML diagrams that are produced far, would form part of this documentation. In this documentation, you would find the following information:

- Specification of the product function.

- A complete description of what the system should do
- The performance requirements it must meet
- Specific details of the operator/system interaction
- Specification of the system's interface with the external environment
- Procedures for error handling and diagnostics
- Constraints on the design and on the development project
- Goals to aim for in the design

The specification of the required behaviour should include:

- Identification of the microprocessor tasks
- Description of how the product as a whole is to interact with its environment, particularly for products with a substantial human interaction

A useful way of providing the specification on how the product is to interact with its environment is to produce a USER'S MANUAL for the yet-to-be designed product.

The start of the system design phase of a project marks an important transition for design engineers as they must make decisions without a outside help.

8.6.1 Decision making in system design

This involves making important decisions on:

- Build or buy ready made parts
- Choice of microprocessor
- Hardware or software

8.6.2 Build or Buy

To build or design from the scratch (i.e. build) may

- not work first time
- need modification

Commercially available controller/microprocessor boards or cards having been proven would not suffer from those problems above.

Another advantage is that cost and performance is known with a high degree of confidence at the beginning of the project as compared to overestimating our own abilities to design and build the required part. Therefore, it is good practice for a beginner to use such commercially available systems as much as possible in early design.

8.6.3 Choice of Microprocessor

Getting the right processor for a new project can give a strong market advantage, but using the wrong chip can weaken an architecture, prolong a design cycle or cripple a product. There are considerations for first timers and those already using a processor. The choice is always important and there are many alternatives. Cost is the overriding factor. But there are one time costs like development and recurring production costs. Some factors to consider are:

- Software availability. This refers to the code used in your products. Is there a large source for code, or old code be reused? Software comprises up to 70% of one time engineering costs.
- Readily available in quantity including reliable second sources (other companies make the part). For long term production, this can be important.
- Experience of others can be used. Training for a new processor is a long and costly affair.

8.6.3.2 Development Tools

Development time which includes programming, is expensive. In order to reduce this, microprocessor vendors have provided information very quickly. This includes data sheets and programmer's guides. Related to this is the architecture of the processor. We also need to answer questions like:

Does it have the features to do the task?
Do we have adequate development time to overcome deficiencies in the part?

Sometimes, the vendors also provide some development support in the form of ready-made boards (including processor, memory, I/O ports). Similar to these are evaluation kits which have more hardware and facilities for software development, like including a monitor program to download and execute user programs through a serial port, and being able to modify portions of memory. These kits also allow for additional hardware to be added on easily by providing extra connectors and sometimes a prototyping area.

Software development tools should be adequate. Assemblers, compilers with simulators and other debugging tools should make the task of programming easier.

This is especially important if the microprocessor's architecture is inadequate in many areas.

Emulator support is of the highest priority. An emulator makes hardware debugging seem like software debugging.

8.6.3.4 Processor Capability

The processor should be fast enough to support the application at hand. This is especially true for real time applications. The addressing space should be enough for program and data.

8.6.4 Software vs Hardware

Processors can be programmed to perform a wide variety of tasks. Engineers are finding that more and more, their task is to utilize available processor hardware by writing the necessary software. However, there is still a need for some electronic hardware design in the interface.

8.6.4.2 Software/Hardware Trade-Off

In general, there would be a number of tasks which can be achieved by means of hardware or software, especially those that interact with external environment and devices. For example, a serial port may be written by manipulating some port pins, or it may be purchased as a hardware component. Not surprisingly, the deciding factor is cost. This can be seen as:

Unit Cost = Component Cost + Production Cost + Development Cost / Total No. of Units

Hardware Implementation
Has an associated component cost but there is

- less development effort
- higher processing speed as the processor does not have to work on this

Software Implementation
No component cost

- may involve large amount of development effort
- In most cases, operation is much slower

8.7 System Design

The method that was introduced in the section on system analysis is also a method for system design. UML is a modern technique of system design and it has been specified as the documentation to be used for all systems proposed in project tenders submitted to the British government.

8.7.1 The Unified Modeling Language

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the features of systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. We mentioned it earlier in System Analysis.

UML represents an important part of system development process. UML uses mostly graphical notations to express the design of system projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of systems.

The primary goals in the design of the UML were:

1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Support higher-level development concepts such as collaborations, frameworks, patterns and components.
6. Integrate best practices.

8.7.1.2 The types of UML diagrams

UML diagrams are designed to let developers and customers view a system from a different perspective and in varying degrees of abstraction. There are altogether six diagrams to be drawn for a UML conformant documentation. In this module, we only need you to draw two - the interaction and the *use case* diagrams.

The UML *use case* diagrams display the relationship among actors (users) and *use cases*. This terminology comes from computer languages where the *case* statement described an action to be taken by a program that depends on the value of a variable, as in the C language.

To draw this, you would first have to imagine you are a user of the yet-to-be built system and you would then come up with the ways offered to you by the system that allows you to interact with the system.

Interaction Diagrams consist of:

a. Sequence Diagram: these diagrams display the time sequence of the objects participating in the interaction. These consist of the vertical dimension (time) and horizontal dimension (different objects).
b. Collaboration Diagram: these diagrams display an interaction organized around the objects and their links to one another.

In this course, our primary focus would be use case and *sequential* interaction diagrams. After the diagrams are drawn, we implement the task to be executed using a combination of hardware and software that the computer has to do. An example is given at the end of the chapter.

8.7.1.4 Hardware Testing

Should be tested on each sub-system basis, using simple programs designed to fully test out the hardware. The test programs should be kept as simple as possible to eliminate the possibility of software errors. This assumes the basic hardware is working correctly.

Modern instruments have been devised to cope with the short-comings of the simple test techniques mentioned above. These are Logic Analyzers and In-Circuit Emulators.

8.7.1.6 Software Testing

After the software code for each subsystem has been written and converted into machine code, it must be tested to verify that it functions correctly. This requires a 'protected' environment to test the software. This allows us to control and monitor the inputs and outputs of the overall system precisely.

For example the environment provided by the monitor or debugger utility program in a microprocessor development system where the performance of the system being tested can be examined in isolation.

All possible conditions going through the control structure is tested. For example, "IF" type of instructions should be tested for both conditions. This is so that all instructions will be checked and do not cause errors later in the project. The aim should be to minimize the number of untested software codes which have to be checked during system integration.

Software that depends heavily on special features provided by the actual hardware should be tested right at the very beginning

8.8 System Integration

System Integration is the process of bringing together individually designed and tested subsystems to form the full system.

- This normally begins by building the whole system from the bottom upwards and implementing the tasks as illustrated in the interaction diagrams.

- Extensively testing the tasks that needs to be performed by the system to verify that no problems have been introduced by linking sub-systems together

Ideally, if modular design has been rigorously applied with well defined interfaces specified, then system integration should be straight forward. However even though we have been careful with our initial documentation, there may still be some inconsistencies and these will not appear until the system integration phase. These are often caused by focussing only on one aspect of the design and neglecting interfacing issues.

If integration is done properly, the complete system can be simply implemented as a series of subroutine calls within a main loop.

8.8.1 Problem resolution

As we are testing the system, we need to check if the system carries out every task as required and it is during this phase that we find inconsistencies. These bugs must be resolved if the system is going to work. It is therefore essential to have access to the initial documentation, and to update it with the lists of bugs found in a structured manner.

If we try to patch up inconsistencies by an ad hoc method, we are likely to get into more and more trouble, particularly if we do it in an undocumented manner.

Although it may appear to be slower, it is more cost effective in the long run to go right back to a task (the one with the bugs) to find out where we went wrong, and then make modifications which are consistent throughout the system.

When the lists have been more or less resolved, we have created a prototype of required system.

Prototype
A prototype is the initial version of the system. There are two types and are used to:

- Demonstrate the validity of the system design
- Show what the system will ultimately look like. We should be very careful about satisfying other needs, such as outward appearance or physical construction. These should only be done if it helps to confirm that the original design concept works.

System Prototype Without Hardware
The development system used as hardware, where input is a keyboard and output on a display.

Once the design has been proved, hardware can then be bought or constructed to produce a second prototype with different goals.

Hardware Prototype
In general, the prototype system is separated from the development system. Software is transferred to the prototype hardware by:

- EPROM
- In-Circuit Emulator (ICE)

8.9 Production

This marks the transition from a prototype to a manufactured product. It may involve changes in original design for the following reasons:

- When product is manufactured in quantity, assembly cost becomes a major factor in the price

- Need to ensure that the system can be assembled and tested by relatively unskilled staff
- Require documentation such as test specifications, service manuals, and user's manuals

8.9.1 Testing of Products

Different from testing during development stage

- Need to perform fault-finding relatively quickly by the use of special test jigs and/or sophisticated test instruments

- Test procedures must be devised

8.10 SAMPLE DESIGN

The following is an example of how we apply UML to a design. Note that there is not just one correct answer. As in all design activity, the designer will justify the choices made in meeting the customer requirements.

> You are required to design a microprocessor based egg timer. This device will show the egg type and boil it for a specified period of time depending on the egg type. It will sound an alarm to signal when it finishes and show a message as well. It must be waterproof.
>
> Perform System Analysis and System Design using UML. Draw a case and interaction diagram of the proposed system. Develop two of the sequential diagrams that you will use.

First, identify the subsystems in the proposed system. Note that we do not need a temperature sensor because the boiling temperature of water is constant. A convenient breakdown of the systems into subsystems would be as follows:

- microprocessor
- alarm
- timer

- LCD
- heating coil
- keypad

Then we draw a UML *use case* diagram. A UML *use case* diagram is another way of expressing user requirements. The diagram helps in uncovering user requirements that may not have been stated earlier on.

8.10.1 Use case diagram

Looking at our example from the user's perspective, the user uses the egg timer for cooking the egg. So a *use case* diagram that a designer would come up with is:

The action marked "cook egg" is a function offered by the system. You might wonder what is the purpose of such a diagram. The process of drawing such a diagram helps you, the designer to identify functions that the system must propose.

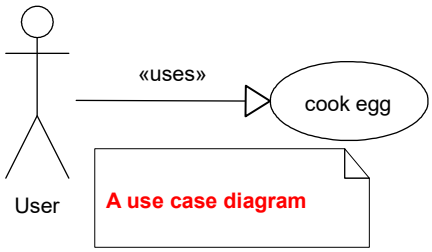


Fig 8.2 Use Case Diagram

Case diagrams are helpful in revealing requirements and planning the project. During the initial stage of a project, most *use cases* should be defined, but as the project continues more functions might become visible as you explore how the system interacts with the user. That is, you will able be to uncover functions that the system should have, but was not explicitly written. In our example, to cook an egg needs to include steps to "choose egg type" before we start to "cook egg".

A general method that one uses to come up with a use diagram is to list a sequence of steps a user might take in order to complete an action. For example:

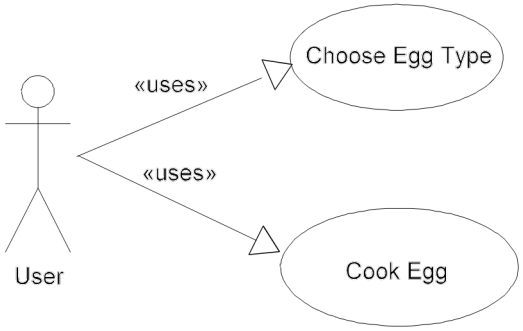


Fig 8.3 Developing the use case diagram

From this simple diagram the requirements of the egg cooker system can easily be derived. The system will need to be able to perform actions for all of the use cases listed. As the project progresses other use cases might appear and this diagram can easily be expanded until a complete description of the egg cooking system is derived capturing all of the requirements that the system will need to perform.

8.10.2 Interaction diagram

These diagrams model the behaviour of use cases by describing the way groups of objects interact to complete the task. The two kinds of interaction diagrams are sequence and collaboration diagrams.

Interaction diagrams are used when you want to model the behaviour of several subsystems in a *use case*. They demonstrate how the sub-system collaborates with others. However, interaction diagrams do not give an in-depth representation of the behaviour. If you want to see what a specific object is doing for several *use cases* we should use a state diagram - which is another component of UML. To see a particular behaviour over many *use cases* or threads use an activity diagram - which is yet another component of UML.

To start drawing an interaction diagram, first imagine how you would interact with the system. In our example, you - the user, would select the egg type using the key type and you would represent the interaction as in the left figure below and note how the subsystem key pad is represented as:

Developing Use case - "Select Egg Type"

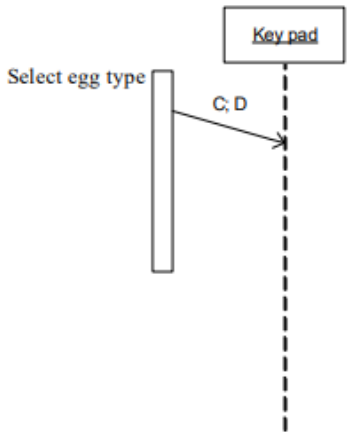


Fig 8.4 Starting Interaction Diagram

Fig 8.5 Part of Interaction Diagram

In like fashion, the other sub-systems will have similar diagrams. The diagram shown below is often used to represent the first interaction with the system in question. In this

case, the first contact between the system and you, the user is the selection of the egg type for cooking by pressing either C(hicken) or D(uck):

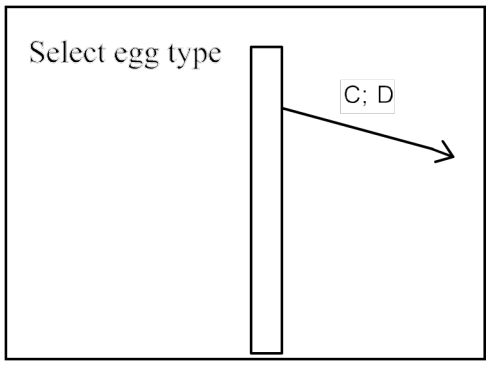


Fig 8.6 Interaction Diagram for "Select Egg Type": Step 1

To develop the interaction diagram further, after the keypad is contacted, the microprocessor would read the key entered, displays the egg type and we have the following diagram:

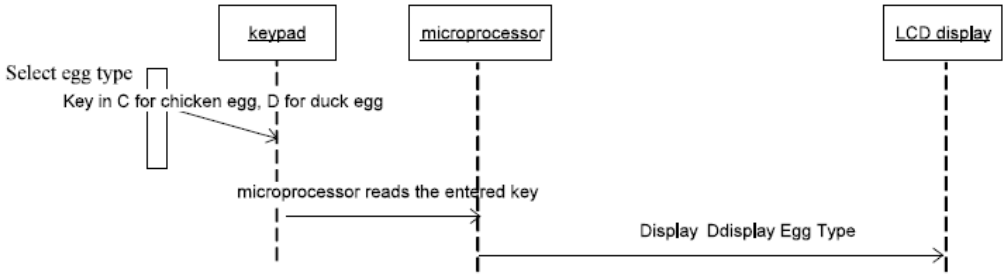


Fig 8.7 Interaction Diagram for "Select Egg Type": Step 2

Developing Use case - "Cook Egg"

In developing this *use case*, we see that once the microprocessor reads the key entered and processes it, it activates the LCD display to show the egg type selected, the timer to start

running for a certain period of time depending on the egg type selected and lastly to activate the heater coil.

After the time *t* is reached, the timer would then send a signal to the microprocessor, telling it that the time is up; the microprocessor would then deactivate the heater coil and send a signal to the LCD display telling it to display the message "egg cooked".

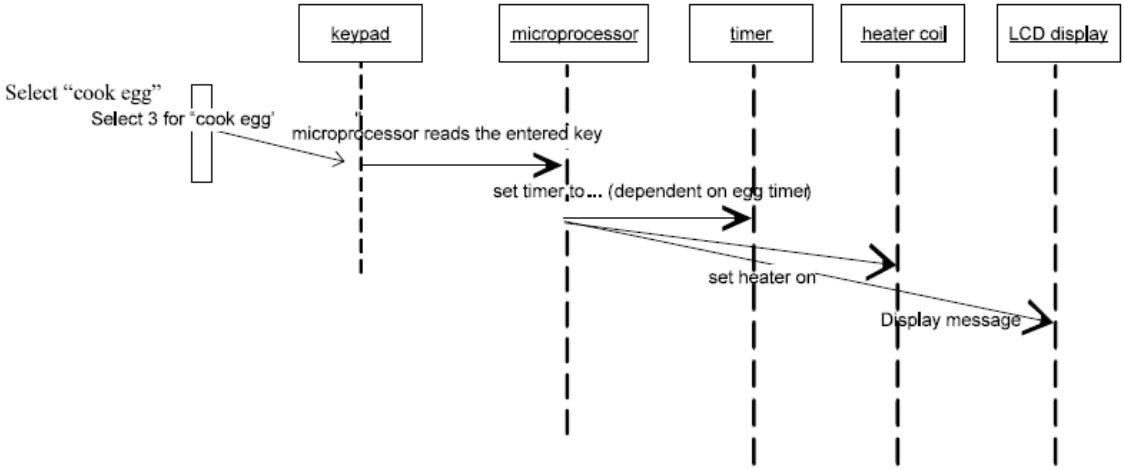


Fig 8.8 Interaction Diagram for "cook egg" step 1

After the egg is considered cooked, we need to activate the alarm. The microprocessor would then reactivate the timer for a much shorter period of time. Then the microprocessor activates the alarm. When the timer's time runs out, the timer interrupts the microprocessor and the processor then deactivates the alarm. Finally, the microprocessor would activate the LCD to redisplay the initial menu.

The complete Interaction Diagram for "cook egg" would look like the following:

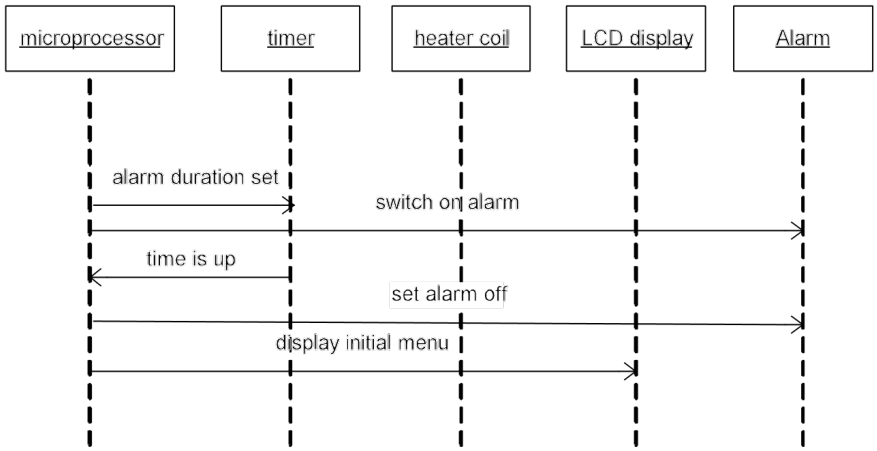


Fig 8.9 Interaction Diagram for "cook egg" : Step 2

As you can see, such a diagram would have helped in implementation and would prove very useful in explaining the implementation of your system to your clients.

Should there be another way in which a user interacts with your system, you would need to repeat the method and produce a similar diagram depicting how the system carry out the task.

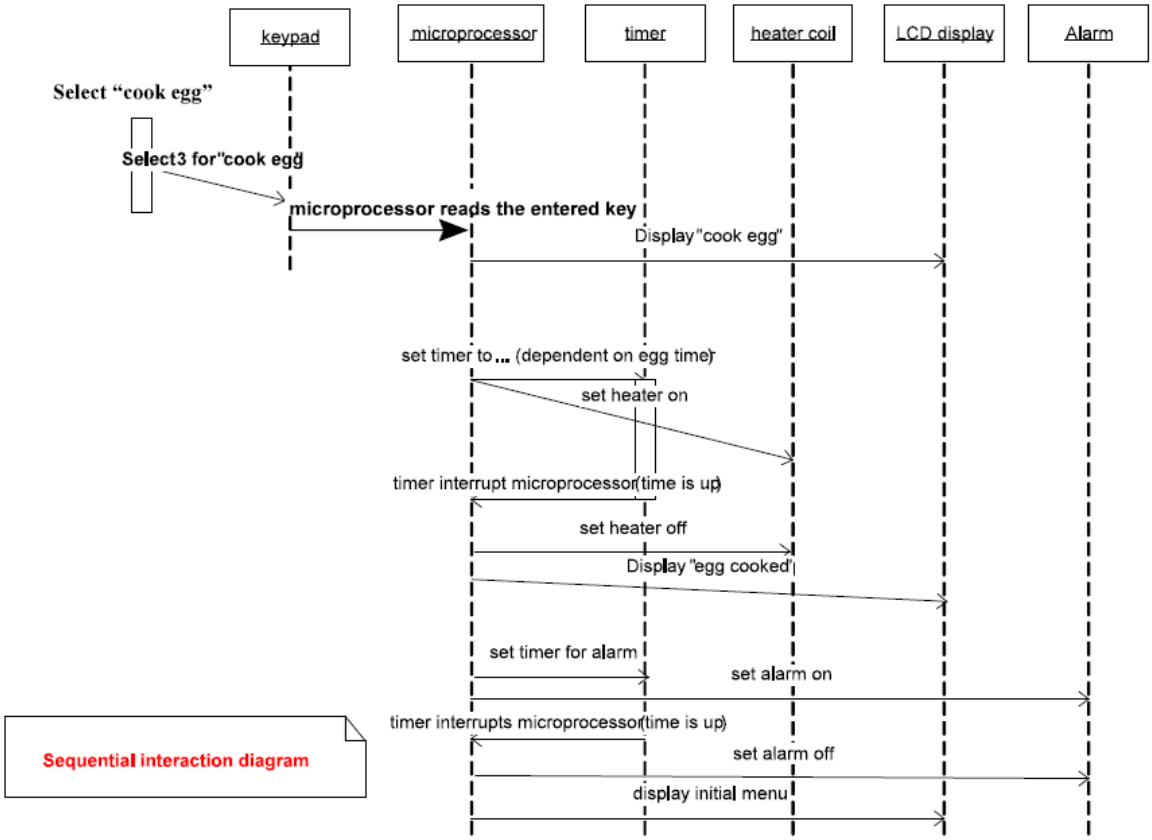


Fig 8.10 Final Sequential Interaction Diagram for "cook egg"

9 Graphic Display Technology

9.1 Introduction

The human visual system responds to pictures more readily than words. Also, certain pictures may have universal meaning, so that communication across different languages is simplified. However the user interface of early computers was text based, due to cost and performance constraints. Graphical user interfaces for computers were developed in the 1970's but were expensive and required a lot of computing power. Today's computers provide high computing power at low cost. The demand for graphical interfaces to various equipment have driven down the cost of graphical interface devices. It is now quite feasible to have a low cost embedded system with graphical interface as can be seen in the various Portable Entertainment Devices available today. Earlier technologies worked with analogue video signals but currently, all-digital data video is coming commonplace. This chapter examines the hardware considerations in advanced embedded systems like the SoC's commonly available.

9.2 Considerations in graphical displays

We live in an age of mobile information technology which relies heavily on the visualization of multimedia data and user input. These tasks are often performed by high-resolution electronic displays equipped with a touch screen or a user input device like a keypad. Choosing a big or a small display, connecting it via the HDMI connector or via the GPIO pins, needing touch capabilities or not, these are only some of the decisions one will have to take depending on the end goal. Ultimately it will be a trade-off with what will the project requires and what's left on the platform to connect and drive a display.

9.3 The nature of a computer image

Because of the digital nature of computers, an image was constructed in the form of a grid. Each cell of a grid would be part of a picture, so it was a picture element or 'pixel'. Each pixel has a colour value, which also includes black for the absence of any colour information. Generally, pixels are square in shape. As in television, the number of pixels making up an image is known as the resolution. This is given as the number of horizontal pixels times the number of vertical pixels. The amount of data required to represent the colour is known as the colour depth. The entire digital image has to be stored in a frame buffer which may be part of system memory. The frame buffer data needs to be transferred to DACs which convert the digital data into analogue colour. The amount of data that needed to be computed and moved to provide the display can tax the resources of the hardware.

Most of today's colour displays are based on three primary colours – red (R), green (G) and blue (B). They can be superimposed in various ways so the human eye sees just the actual colour.

Computer images are displayed at a fixed resolution which specifies the number of pixels to be displayed horizontally and vertically. The colour at each pixel is represented by digital data. This quantity is known as the colour depth. Commonly used quantities are 1 bit, 4 bits, 8 bit gray and colour, 24 bit colour, and 32 bit colour. The easiest to store is 24 bit colour, where one pixel has three bytes stored. One each for its component colours. Of course this takes the most space.

The 8 bit colour scheme is a special type of image data storage. Colours can be represented as a combination of three 8 bit quantities. However, not all possible values of all colours (16 million - 2^{24}) are used in an image. It is possible to create a good approximation to the colours in a picture by just using 256 of the most common colours in *that* image. This process of quantization uses only 1/3 of the data needed, which is a great saving in file space. All this can be done by hardware at high speeds, so that the job of finding the most common 256 colours to represent a picture is done quickly. But it must be noted that often, with images like photographs, the effect of reducing the colours results in *posterization* where large patches of a single colour will replace a large area of smoothly changing colours. Thus using 256 colours should be used for artificially generated images like cartoons or animations. Each of the 256 colours are stored in the image as 3 byte entries. This structure is known as a palette. So each pixel of an 8-bit image points to a 3-byte entry in the palette. Because each image is different, a palette has to be stored with *every* image. This is not so with 8 bit gray scale images as each 8 bit image actually represents just the brightness information only.

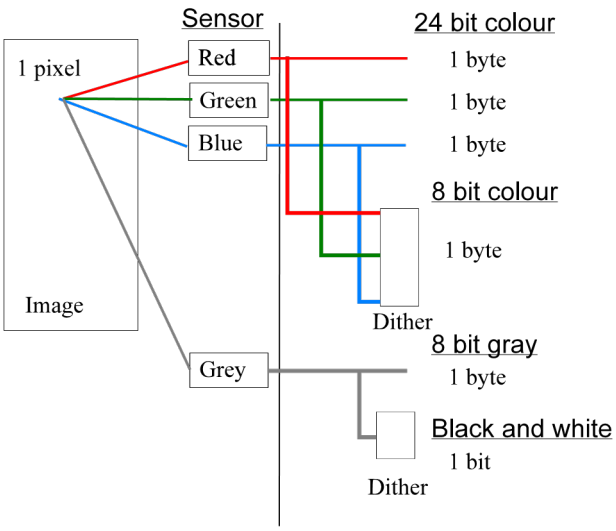


Image formation and colour depth options

Gray colours are often created by a linear combination of the RGB colours. By suitable thresholding of gray values, 1 bit, or black and white images may be obtained.

The following diagram gives an overview of how a pixel may be represented digitally.

To display an 8 bit colour image, the values of the palette are loaded into a hardware *look-up table*. This is used to drive the colour DACs which provide the signal for the appropriate colour.

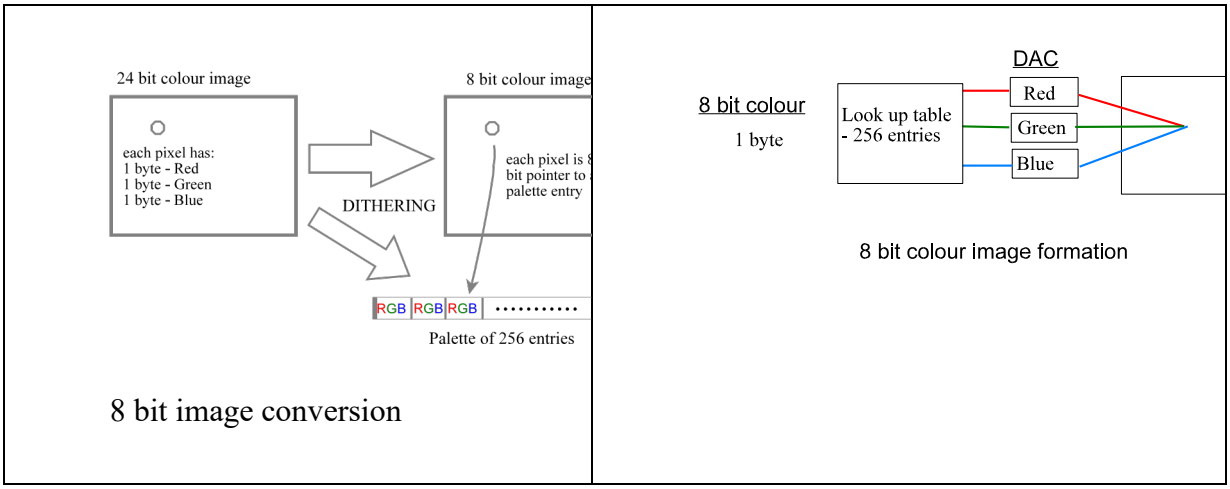


Fig 9.1 8-bit colour image formation

9.3.1 Display technology

There are many different technologies used to build a display. These technologies are typically known by the acronym that refers to the primary electro-optical effect that is used to emit and/or modulate visible light at a pixelated level, e.g CRT, LCD and OLED. This may be further divided into technologies which generate their own light – emissive, or those that modulate light generated separately (e.g via a backlight) – reflective or transmissive. Displays are also referred to as being direct view, virtual or projected to describe the manner in which the image is viewed by the user. These categories are used in Fig. 9.2 to provide an overview of the major technologies:

- Direct view displays - the user looks direct onto a display
- Projection displays - the user sees the image of a microdisplay projected onto a reflective screen
- Microdisplays - the user views an image which is typically displayed near the eye
- 3D displays - displays which can produce three-dimensional images

The different characteristics of emissive, transmissive, and reflective displays are presented in Fig. 9.3. As simplification, the principles are only visualized for black and white, with a summary of the corresponding display features in Table 9.1

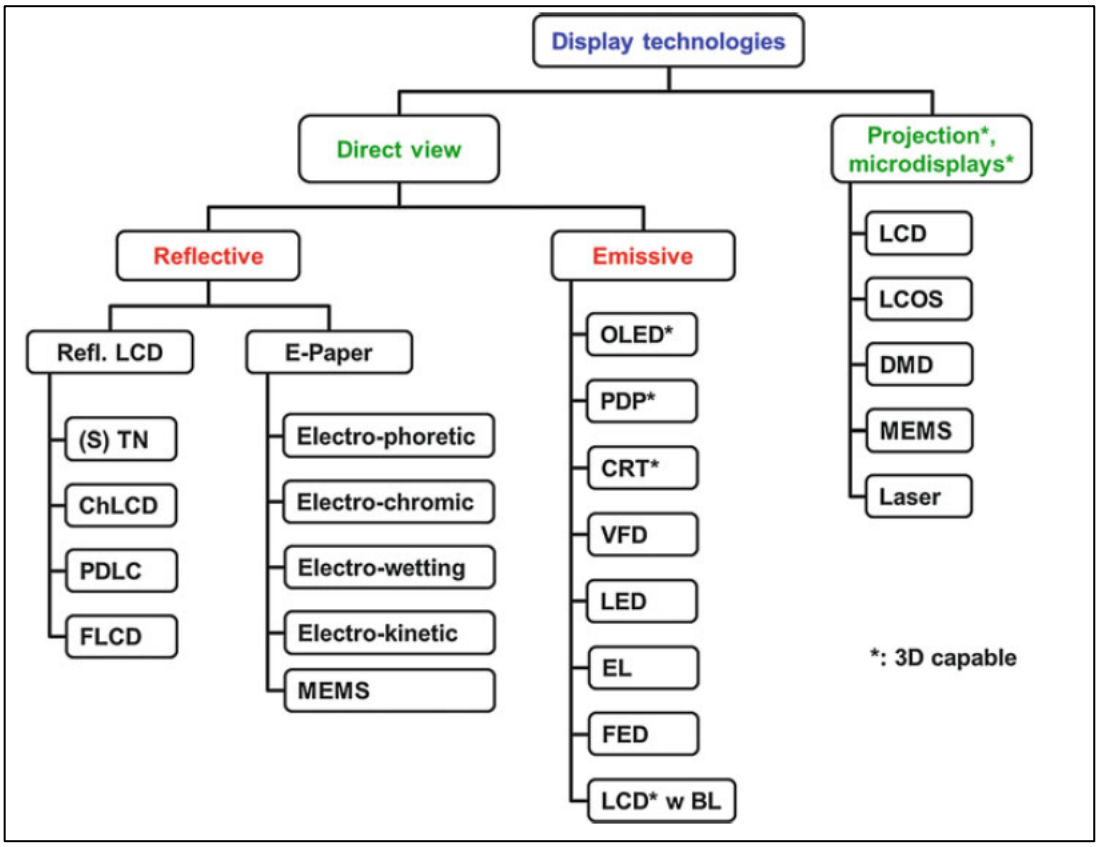


Fig 9.2 Overview of display technologies (see Appendix for definitions of the abbrevations)

- Emissive displays generate light by converting electrical power to light.
- Reflective displays modulate their reflectance, typically via a voltage-controlled effect.
- LCDs are labelled as both reflective (e.g., 7-segment display as used in a watch) and transmissive.

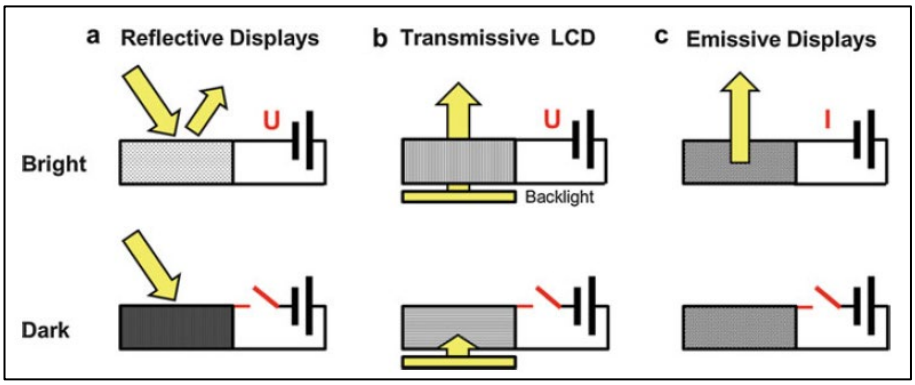


Fig 9.3 Visualization of fundamental electric-to-optic conversion for major flat panel display technologies

Topic	Reflective	Emissive/transmissive
Examples	Segmented LCD Passive matrix LCD E-paper	OLED Plasma Display Panel (PDP) Active matrix colour LCD (AM LCD) with backlight
Typical merits	Low power Sunlight readable	Multimedia Supply chan of AM LCDs
Typical shortcomings	No or limited colour Some with slow response time	Bright light performance High power consumption

Table 9.1 Techology dependant typical display features

- The reflection characteristic of reflective (non-emissive) displays is controlled by a voltage (U, low current, Fig. 9.3a) switching between black (absorption, low reflectance) and white (high reflectance). These displays are therefore readable in bright light but must be illuminated when dark. As ambient light is utilized, the power consumption is extremely low ("green displays"). Examples are segmented monochrome LCDs and e-paper displays. The latter ones are mostly bistable, which means that electrical power is only needed when a pixel should change its status. This is very significant in terms of power consumption, which facilitates very long battery life for e-readers based on this technology.
- Transmissive display (Fig. 9.3b) is the term used for color LCDs (mostly active matrix) which are equipped with a bright backlight. A voltage changes the transmission between a low (black) and a high value (white) of the liquid crystal layer (cell) to modulate the light from the backlight subsystem. So the power consumption of the basic liquid crystal cell is low but that of the display is high due to the necessary backlight. Because of the reliance upon an integrated light sources, transmissive LCDs can also be referred to as emissive displays, and even as "LED" displays, which actually only refers to the backlight.
- Emissive displays (Fig. 9.3c) generate light by converting electrical power into visible light. Note that "I" is used as the current symbol here, since these are often current driven, and, relative to reflective technologies, higher powers are typically required. For emissive display, the power consumption also depends on the actual data shown; hence, a black background is preferable in this context. Examples are OLED, PDP, CRT, VFD, FED, EL, and LED.
- Transmissive and emissive displays typically provide a high-quality optical performance under indoor ambient conditions, but their outdoor use is limited due to the need for a very high luminance to provide sufficient contrast. This requires higher power consumption, hence reducing battery life and impacting on longevity of the display system due to heat generation.

It is evident that no single display technology exists which is suitable for all applications. A key decision point is the choice between low-power monochrome displays and multimedia screens consuming a relatively large amount of power.

9.3.2 Display Driving Principles

After introducing pixels earlier, the next step toward understanding how a display works is to consider how the pixels are electrically driven, e.g., setting a gray level or switching between gray levels (the simplest case is black ↔ white). The two fundamental approaches are shown in Fig. 9.4 – direct drive (a) and matrix drive (b). For direct drive all segments (typically used instead of "pixel" for this case) are directly connected to the driving electronics. So there is full control over the voltage and the waveform of any segment. The arrangement and the shape of the segments can be individually designed for the application to be, for example, bars or icons, with the geometry defined by electrode structuring.

. Matrix drive can be classified into passive matrix drive and active matrix drive.

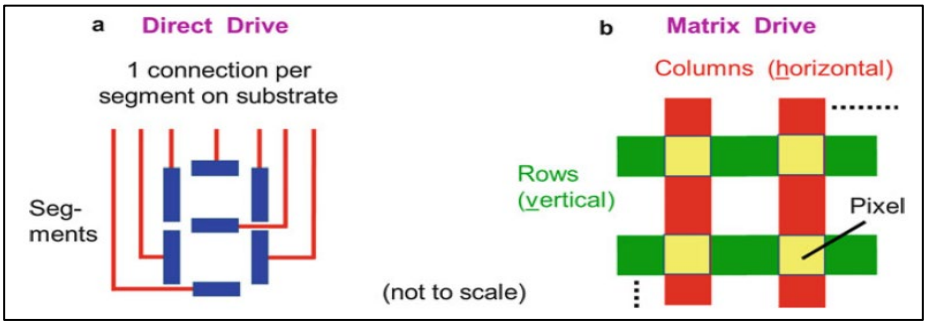


Fig. 9.4 Fundamental driving principles of pixels: Direct (a) and matrix (b) drive

9.3.2.1 Passive matrix

In a *passive matrix* display (Fig. 9.5a), pixels are addressed row by row, this is called time multiplexing. That means that all pixels on row 1 are updated first, then all pixels on row 2, etc meaning that for a display with three rows, each row is only addressed ⅓ of the total time. On retro displays (e.g CRT), it is sometimes possible to see this effect as a continuous sweeping across the screen. For LCDs, this reduces the contrast of the display which, in turn, limits the total number of rows possible. This method is often called *multiplex driving* for segmented displays. Passive matrix drive is a cost-effective method to drive displays as it doesn't require any additional hardware. However, just a few display technologies have the characteristics required for passive matrix drive.

9.3.2.2 Active matrix

The sophisticated display, that you most likely are looking at right now, is based on active matrix (Fig 9.5b) technology. In an active matrix, each pixel contains at least one transistor. More often, there are multiple transistors and capacitors. In an OLED for example, each pixel contains a quite sophisticated circuit with 5-10 transistors. By adding transistors to the pixel, it can more easily be controlled. This is partly because transistors offer a threshold voltage which is an important feature for a display matrix to function properly. A capacitor, on the other hand, functions as an energy storage when the pixel is not addressed. In this way, all pixels can maintain their state even for a large number of rows. The Apple iMac display, for example, can in this way achieve 2880 rows without a problem. The drawback with active matrix is the high price point since the

fabrication requires expensive deposition processes. For that reason, active matrix is mainly suitable for high-end displays.

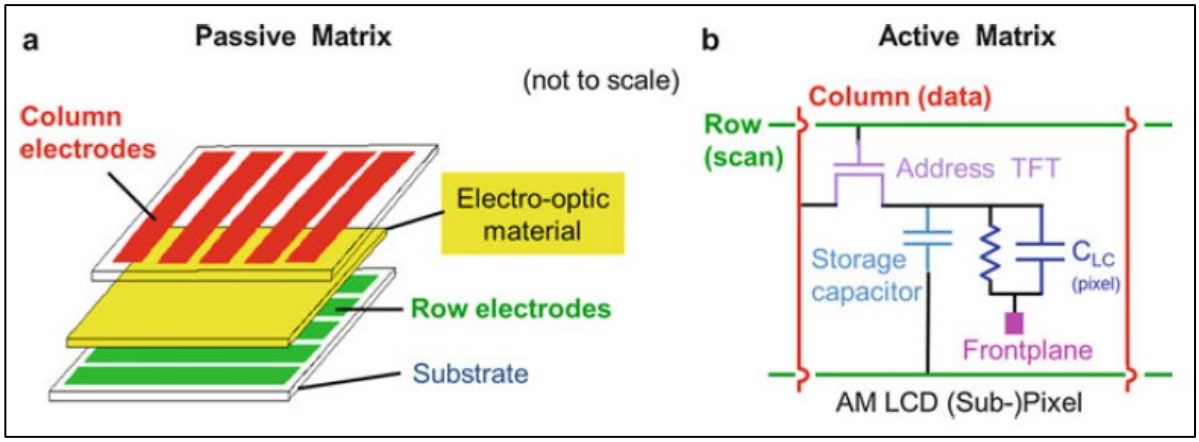


Fig. 9.5 Schematic drawing of passive (a) and active (b) matrix driving

9.4 Types of interfaces of display devices

Current embedded systems can be connected to a display in several ways, depending on the model. We look at some of the interface that is typically used in a project.

Composite Video

Despite the great success it has received the Raspberry Pi has humble origins and at its heart it was designed to be inexpensive. It was thought that providing Composite Video would have allowed in many countries that still rely on old TV set to use the Raspberry Pi without the need to purchase a dedicated monitor. A composite video connection transmits a basic analog video signal between devices. It sends standard-definition video only - and does not carry an audio signal. If you use a composite video signal for the picture, then you will need a separate connection to hear the audio.

DSI

DSI (Display serial interface) is a high-speed serial interface based on a number of (1GBits) data lanes. The total voltage swing of the data lines is only 200mV which is extremely small, and therefore the term low voltage differential signalling (LVDS) reflects this. The electromagnetic noise created and power consumed is very low.

HDMI

HDMI (High-Definition Multimedia Interface) is used for transmitting uncompressed video or digital audio data to the Computer Monitor, Digital TV, etc. Generally, this HDMI port helps to connect your current embedded design to the Digital television. In the vast majority of cases, simply plugging your HDMI-equipped monitor into for example, a Raspberry Pi using a standard HDMI cable will automatically lead to the Pi using the best resolution the monitor supports. You should connect any HDMI cable before turning on the Raspberry Pi.

GPIO

GPIO is the acronym for General Purpose Input Output and very often the GPIO pins are multiplexed with other functions so they need to be set up before use. So for example, you can reconfigure each pin as either an input or output but also deem them to provide a different type of interface for the various types of displays and boards in general. You can find displays that use DPI, SPI, I2C and even UART or an ad-hoc interface. Some Interfaces like DPI are particularly fast but will require lots of GPIO to interface with the screen, others like SPI and I2C need very few pins but won't be as fast as DPI. Ultimately choosing between them comes down to your projects requirements.

- DPI
  DPI stands for Display Parallel Interface (or Parallel Display interface). It allows you to use very cheap displays by driving them manually. However, it may not be right for every project.

  Pros:
  Very fast, easily driving display at 60hz.
  No complicated interface hardware.
  Pixel perfect output. Digital, not analog.
  Easy to understand protocol.
  No bulky connectors.
  Very inexpensive.
  Cons:
  Uses a lot of GPIO pins
  Ribbon cables break easily if you're not careful.
  Short range.

  The DPI interface allows displays to be attached to the Raspberry Pi GPIO either in RGB24 (8 bits for red, green and blue) or RGB666 (6 bits per colour) or RGB565 (5 bits red, 6 green, and 5 blue). If we wanted to drive the display at full 24 bit true color (RGB24) we would need 8 pins each for red, green, and blue as well as pins for signals like hsync, vsync, clock, and display enable. This is a total of 28 pins. The way to reduce pin count is to simply omit the two least-significant-bits on each color bus, giving us 18 bit high color (RGB666). Another option is to use RGB565. Obviously we lose some colour information but it's not as bad as one would expect. This is one way the SMI interface described in an earlier chapter can be used.

- SPI/I2C (or I^2C)
  The Serial Peripheral Interface (SPI) and Inter-Integrated-Circuit bus (I2C) are serial interface on the GPIO. With SPI the amount of pins required is much less than DPI, it only requires 4-5 pins. More than one device can be connected to SPI but it's generally not used for more than one device.

  I2C requires few pins, in fact it only needs 2 which makes this interface very light in terms of GPIO usage. One additional advantage is that it's very easy to connect more than one device to I2C bus making this option quite versatile.

One of the main downsides of these technologies is that you will end up with fairly small displays and very low resolutions.

- UART
  UART stands for Universal Asynchronous Receiver/Transmitter. A UART's main purpose is to transmit and receive serial data. Almost all microcontrollers have dedicated UART hardware built into their architecture. The main reason for integrating the UART hardware into microcontrollers is that it is a serial communication and requires only two wires (and ground) for communication.

Appendix:

ChLCD	Cholesteric Liquid Crystal Display
CRT	Cathode Ray Tube
DMD	Dot Matrix Display
EL	Electroluminescent
FED	Field Emission Display
FLCD	Ferroelectric Liquid Crystal Display
LCD w BL	LCD with Backlight
LCoS	Liquid Crystal on Silicon
MEMS	Microelectromechanical Systems
OLED	Organic LED
PDLC	Polymer-Dispersed Liquid Crystals
PDP	Plasma Display Panel
TN	Twisted Nematics
VFD	Vacuum Fluorescent Display

10 Graphical User Interfaces

10.1 Introduction

Have you ever had difficulty in finding your way round a website, using the latest piece of software, or operating your mobile telephone? Why is so much computer-based technology difficult to use?

A vast array of computer-based systems surrounds us: stand-alone and networked personal computer systems, websites, safety-critical systems and embedded systems such as mobile telephones and microwave ovens. They influence every aspect of our lives, both at work and in our leisure time. Their effectiveness depends primarily on the user interface (UI), which allows the user to interact with the computer system built into them. Poor UI design can have serious consequences. How many of us would want to fly in an aeroplane whose pilot was unsure what all the knobs and dials did? For organisations, poorly designed UIs can increase training costs and reduce productivity. For the user, they can cause frustration and inefficiency. How often have you moved to another website because you cannot find the information you need?

10.2 General User Interface Design

User Interface Design is often associated with software interface and is frequently referred to as Human-Computer Interface or HCI. However, User Interface Design must be considered wherever users interact with controls or displays. The application of interface design is commonly found, these include products such as: a simple watch, a DVD player, an aircraft cockpit, a software program, etc. In many cases, good technology is not readily accepted because the product is not easy or efficient to use. A product's usability, acceptance, and marketability are often dependent on the user feeling that it is easy to learn and use. User Interface Design increases the intuitiveness, efficiency, and comfort level with a product, which translates into product acceptance and use. You need both good technology and usability for a successful product. Many technological innovations rely upon User Interface Design to elevate their technical complexity to a usable product. Technology alone may not win user acceptance and subsequent marketability. The User Experience, or how the user experiences the end product, is the key to acceptance. And that is where User Interface Design enters the design process.
When applied to computer software, User Interface Design is also known as Human-Computer Interaction or HCI. While people often think of Interface Design in terms of computers, it also refers to many products where the user interacts with controls or displays. Military aircraft, vehicles, airports, audio equipment, and computer peripherals, are a few products that extensively apply User Interface Design.

The importance of good User Interface Design can be the difference between product acceptance and rejection in the marketplace. If end-users feel it is not easy to learn, not easy to

use, or too cumbersome, an otherwise excellent product could fail. Good User Interface Design can make a product easy to understand and use, which results in greater user acceptance.

10.2.1 User-Interface in Embedded Systems

Embedded systems involve close interaction with real-world physical components and mechanisms. This interaction takes the form of control, coordination, or monitoring of actual electronic or electromechanical devices. What stands out is both the importance of real-world objects and the sequence of events or transactions taking place between the embedded software, external physical objects, and their users.
An embedded system for a given application must ultimately exhibit the same effective behaviour and external characteristics regardless of how the internal program is realized. These features and capabilities can be embodied into which their performance must conform to requirements.

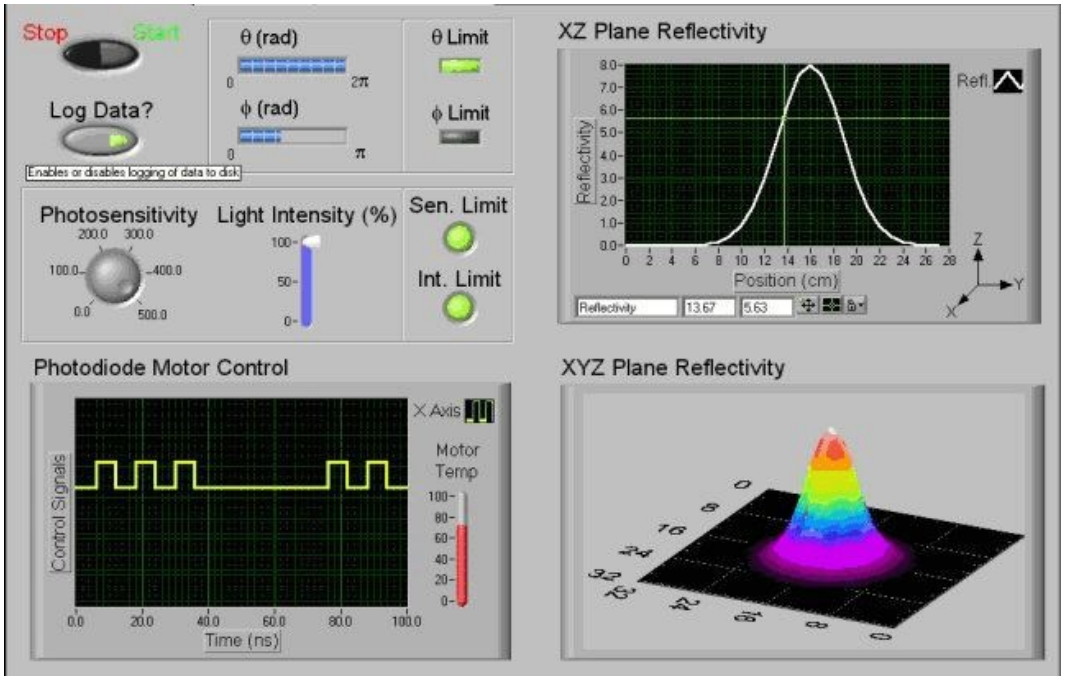


Fig 10.1 An example for the screen display of User Interface Design

10.3 Principles to consider for effective User Interface Design

This section describes some general factors which determine an effective user interface design. By this we mean how well the product interface helps the user accomplish a task. The ease of use of the interface plays a great part in the productivity of the user. After all the product is meant to help, not hinder the user! Principles should always be in the mind of the designer as they work on an interface. Guidelines will help the designer realise a principle.

10.3.1 Consistency

The importance of maintaining strict consistency varies according to the level of the task required. It is useful to be aware of when to apply them. Now, it is just important to be visually inconsistent when things must act differently as it is to be visually consistent when things act the same. We should avoid uniformity just for its own sake. Make objects consistent with their behavior. Make objects that act differently look different from the rest. The most important consistency is consistency with user expectations. The only way to find out user expectations is to do user testing, which no amount of study and debate will substitute.

10.3.2 User Efficiency

Look at the user's productivity, not the computer's. People cost a lot more money than machines, and while it might appear that increasing machine productivity must result in increasing human productivity, the opposite is often true. In judging the efficiency of a system, look beyond just the efficiency of the machine. As an example, which of the following takes less time? Heating water in a microwave for one minute ten seconds or one minute eleven seconds? Of course one minute ten seconds is obviously faster. But the user must press the one key twice, then visually locate the zero key, move the finger into place over it, and press it once. In the second case, the user just presses the same key–the one key–three times. It typically takes more than one second to locate the zero key.

Other factors beyond speed make the “111" solution more efficient. Seeking out a different key not only takes time, it requires a fairly high level of thinking and care. While the processing is underway, the main task the user was involved with–cooking their meal–must be set aside. The longer it is set aside, the longer it will take to reacquire it. In addition, the user who adopts the expedient of using repeating digits for microwave cooking faces fewer decisions.

A good example of this is Fitts' Law which says that “The time to acquire a target is a function of the distance to and size of the target”. In other words, an often used object or action should be large and easily accessible - by menu or mouse.

10.3.3 Using Human Interface Objects

Human Interface Objects are representations of objects that humans can relate to.

i) They can be seen, heard, touched, or otherwise perceived.
ii) Visual interface objects are quite familiar in graphic user interfaces, although those using other human senses are available.

iii) Since they relate to standard objects, they must have a standard way of interacting with it.

iv) They must also have standard resulting behaviours and not produce surprises.

Some familiar objects include folders, documents, and the trashcan. Thus, icons are typically used to represent them, pictorially.

Metaphors

As an extension to this concept, metaphors are representations of system objects that correspond to the familiar real world objects, so that handling them would not require extra training, as they operate like actual objects. For example, Windows has an object called a briefcase. Like a real-world briefcase, its purpose is to help make electronic documents more portable. Choosing metaphors well, will enable users to instantly grasp the finest details of the conceptual model. Bring metaphors alive by appealing to people's perceptions–sight, sound and touch as well as triggering their memories.

10.4 Ergonomic Guidelines for User-interface Design

Ergonomics is one major factor that made a big difference for an effective GUI design. The design concept will always have the end user in mind. Good ergonomics makes every effort to ensure a user friendly and comfortable user environment.

The following points are guidelines to good software interface design, which is to help implement the principles discussed. These guidelines apply to the content of screens. In addition to following these guidelines, effective software also necessitates using techniques, such as 'storyboarding', to ensure that the flow of information from screen to screen is logical, follows user expectations, and follows task requirements. These are presented in point form.

10.4.1 Consistency

- certain aspects of an interface should behave in consistent ways at all times for all screens
- terminology should be consistent between screens
- icons should be consistent between screens colours should be consistent between screens of similar function

10.4.2 Simplicity

- break complex tasks into simpler tasks
- break long sequences into separate steps
- keep tasks easy by using icons, words etc.
- use icons/objects that are familiar to the user

 - delay (processing)	 - save	 - zoom

10.4.3 Human Memory Limitations

- organize information into a small number of "chunks"
- try to create short linear sequences of tasks
- don't flash important information onto the screen for brief time periods
- provide cues/navigation aids for the user to know where they are in the software or at what stage they are in an operation
- provide reminders, or warnings as appropriate
- provide ongoing feedback on what is and/or just has happened
- let users recognize information by making it available rather than try to remember it.

10.4.4 Cognitive Directness

This term refers to the mental processes that a user needs to use in order to get a device to perform a function. To make this process more "direct" is to reduce the amount of mental work needed a user uses less effort to accomplish a task. The shortest number of steps taken to perform an operation may not be the most intuitive!

- minimize mental transformations of information (e.g. using 'control-I' to indent a paragraph)
- use meaningful icons/letters
- use appropriate visual cues, such as direction arrows for movement of information
- use 'real-world' metaphors whenever possible (e.g. desktop metaphor, folder metaphor, trash can metaphor etc.)

10.4.5 Feedback

It is useful to identify the types of feedback that can be used. This makes us think more carefully about how to use them. Those discussed below are a typical selection of feedback types. This is especially true for the system is processing data.

For time related considerations:

- Acknowledge all button clicks by visual or aural feedback within 50 milliseconds.
- Display some sort of "waiting" indicator (e.g. hourglass) for any action that will take from 1/2 to 2 seconds.
- Animate the indicator so users won't think the system has "hung".
- Display messages indicating the potential length of the wait for any action that will take longer than 2 seconds.
- Communicate the actual length through an animated progress indicator.
- If possible, show text messages to users informed and entertained while they are waiting for long processes to complete.

- Have the system give a large signal upon return from lengthy (>10 seconds) processes, so that users know when to return to using the system.

For other types of feedback,

- give informative feedback at the appropriate points - in the form of messages
- provide appropriate sensory feedback
- confirm the physical operation you just did (e.g. typed 'help' and 'help' appears on the screen).
- this includes all forms of feedback, such as auditory feedback (e.g. system beeps, mouse click, key clicks etc.)

- provide appropriate semantic feedback – 'semantic means 'meaning' - this type of feedback is meaningful to the action being performed. It is also time consuming to implement as it has to recognize the context of an action. (e.g. when adding up a series of numbers, a running total is shown, or when selecting an object, a set of icons showing possible actions are shown)

- provide appropriate status indicators to show the user the progress with a lengthy operation (e.g. the copy bar when copying files, an hour glass icon when a process is being executed etc.)

10.4.6 System messages

- provide user-centered wording in messages (e.g. "there was a problem in copying the file to your disk" rather than "execution error 159")
- avoid ambiguous messages (e.g. hit 'any' key to continue – there is no 'any' key and there's no need to hit a key, reword to say 'press the return key to continue)
- avoid using threatening or alarming messages (e.g. fatal error, run aborted, kill job, catastrophic error)
- use specific, constructive words in error messages (e.g. avoid general messages such as 'invalid entry' and use specifics such as 'please enter your name')
- make the system 'take the blame' for errors (e.g. "illegal command" versus "unrecognized command")

10.4.7 Anthropomorphization

- don't anthropomorphize (i.e. don't attribute human characteristics to objects)
- avoid the "Have a nice day" messages from your computer. It can come across as patronizing rather than cute.

10.4.8 Modality

- a mode is an interface state where what the user does has different actions than in other states (e.g. changing the shape of the cursor can indicate whether the user is in an editing mode or a browsing mode)

- make user actions easily reversible – use 'undo' commands, but use these sparingly
- allow escape routes from operations

10.4.9 Attention

- Use attention grabbing techniques cautiously (e.g. avoid overusing 'blinks' on webpages, flashing messages, 'you have mail', bold colors etc.)
- don't overuse audio or video
- use colors appropriately and make use of expectations (e.g. don't have an OK button colored red! Use green for OK, yellow for 'caution, and red for 'danger' or 'stop')
- don't use blue for text (hard to read), blue is a good background color
- don't put red text on a blue background
- use high contrast color combinations
- use colors consistently

 - correct or select	 - error or close	 - unexpected action

10.4.10 Display issues

- maintain display inertia – make sure the screen changes little from one screen to the next within a functional task situation
- organize screen complexity - eliminate unnecessary information
- use concise, unambiguous wording for instructions and messages
- use easy to recognize icons, be sensitive to cultural interpretation

10.4.11 Individual differences

- accommodate individual differences in user expertise (from the novice to the computer literate)
- accommodate user preferences by allowing some degree of customization of screen layout, appearance, icons etc.

- allow alternative forms for commands (e.g. key combinations through menu selections or mouse click selection

10.5 User Interface Design for Embedded Systems

The previous sections have discussed interface design in general. Embedded systems are part of a product and there is a close level of interaction between the user. In fact, they may be used on a daily basis by untrained users. Also, these systems may be controlling critical systems. Thus we have to recognize that the user interface is key and we must put users at the center of the design and development process.

If the interfaces are bad, complaints about the controls equipment will surface. Users have to refer back to manuals. To some extent, user may use only a fraction of the capabilities of systems.

When it comes to the user interface, embedded systems differ from "standard" applications development that use the entire computer screen as the primary interaction interface. The product containing the embedded system almost completely determines or constrains the appearance of the interface to the user.

Input devices are limited switches and keypads, outputs to LEDs, buzzers and screens of any sort are difficult to come by. However, there may be a great amount of freedom in the placement these devices subject to constraints of a product. For example, given a certain chassis size, there is quite a lot of freedom in where to place switches and LEDs. But it is very difficult to change these once the positions are fixed.

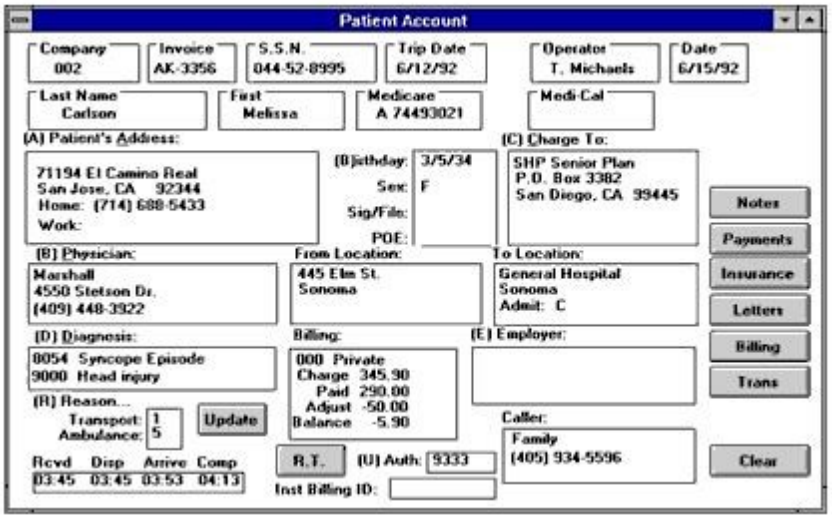
10.5.1 User-Centered Design

This is more than just being "user-oriented"; it makes the end user the focus that informs the entire design process. User-centered design is not about "user friendliness" either, but about making systems that are substantially easier for users to make use of — simply, quickly, and reliably— that make it easier to do things well.

Better user interfaces can be designed if you keep in mind some broad principles about human-machine interaction. Here we will mention a few of the more important ones as they apply to user interfaces for embedded system applications.

Keeping the user informed means that signals or messages to the user must be clear and clearly distinguishable. Sometimes the output devices for embedded systems applications are somewhat simplified. There might be only a set of LEDs or a small LCD panel, but even such simple devices can be used remarkably effectively to communicate a range of information.

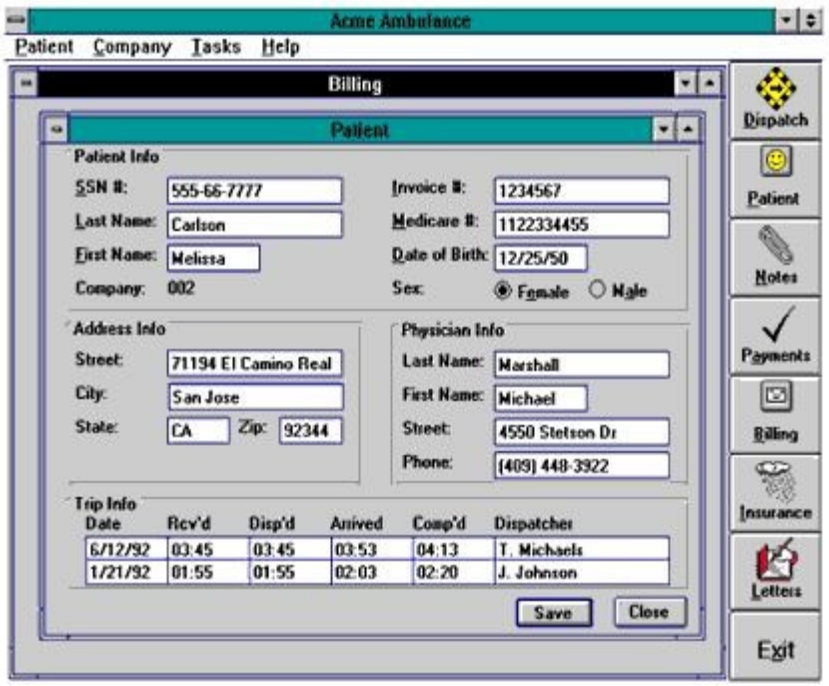
An effective way to get better user interfaces is to take a critical, investigative approach, studying the user interface, studying users, and studying users using the user interface. The following is a GUI design in need of redesign. See if you can identify its purpose. Here is what is wrong with this screen:



In respect to the design guidelines mentioned earlier, some of the problems are:

Consistency - Groupings of data have no pattern, not properly arranged, showing lack of symmetry.
Cognitive directness - Too many buttons, not explained well, no icons. Difficult to know what to do!
Simplicity - there is no indication of the overall tasks and subtasks, don't know where to start.

Here is an improved version of the GUI. It shows a much improved interface for this same application:



By reordering input fields and using icons in the controls, using the interface has become a more pleasant and productive task.

10.5.2 Critical Test

The critical test is to give someone a prototype, or simulation, and tell them what the system is supposed to do. Then let try to use the product without a manual or further instructions. To get the most from these studies, you should video the session, then review the session with the user. Keeping in mind that any problem the user has, however minor, is not a symptom of their ability or inattentiveness, but a problem in the user interface that ought to be examined more closely. Every mistake or misstep they make indicates a probable design flaw.

10.5.3 Usability for GUI in Embedded Devices

Adding a graphical display to your product may allow you to add more features in a smaller space, but it also raises usability issues.

With a non-graphical display, one layout of buttons and displays has to be designed and evaluated. With a graphical user interface (GUI) there is no limit to the number of possible layouts. Making each one user friendly, while remaining consistent with the others, is a big challenge.

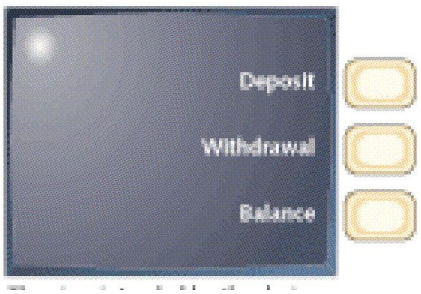


Fig 10.2 Example of GUI for ATM machine

While a GUI has many advantages, it is important to note a couple of the disadvantages.

i) Though a GUI allows a number of different controls on the screen, they all have the same tactile feel when making an input, needing more visual attention as compared to say, a push button as compared to a switch.

ii) Another disadvantage of the GUI is that space does not generally permit the important controls to be permanently visible. A related problem is that if only a GUI is used, it will not be possible to have all of the controls visible at all times. This means that the user may have to explore the interface to find some of the functions.

GUI used with hardware

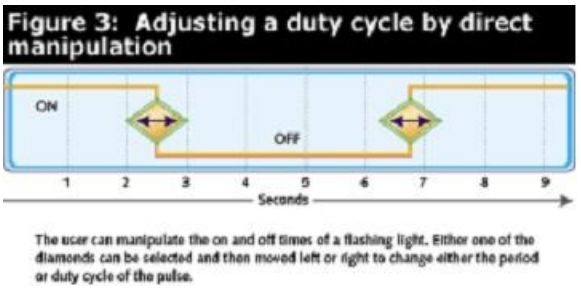
Many embedded products get the best of both worlds by adding a graphics screen to support peripheral information, while the most important user dialog still takes place using custom

controls. The next level of GUI is to provide interactions where the input and output is graphical by nature, not just a set of controls that could have been implemented with mechanical switches, dials, and sliders. For example, instead of outputting a numerical value, the value could be graphed over time giving the user a better sense of the changes within the control process.

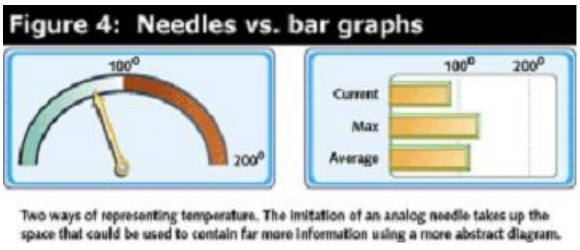
10.5.4 Different types of Interaction in GUI

More precisely, a GUI can be made up of several types of *controls* which the user manipulates, or to display information. In some systems, these controls are also known as *widgets*.

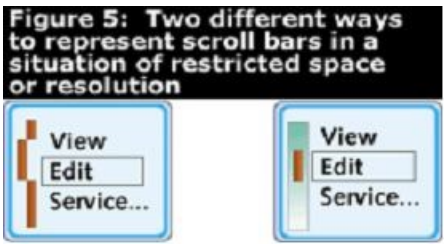
Tick-box Control



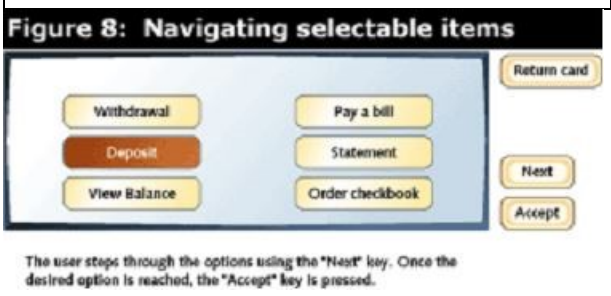
Direct Manipulation



Needles and Bar Graphs



Scroll Bars



Navigational Select Buttons

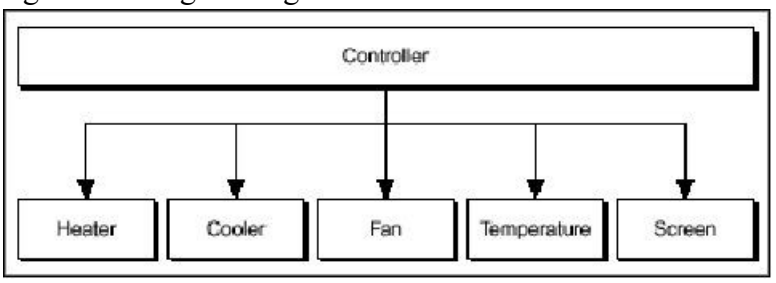
Appendix

CASE STUDY

Description

The embedded system for an air conditioning system. The control system consists of the following subsystems:

Data acquisition board with integrated analog and digital I/O
A 9 in. LCD screen
Heater
Cooler
Fan Temperature Sensor



The heater and cooler are both turned on and off with digital signals. Analog signals are used to inform the heater or the cooler how many degrees the air needs to be heated or cooled, respectively. The fan has five different speeds controlled by digital signals. From the temperature gauge the current temperature is sent by an analog signal.

The relationship between the classes is represented in Universal Modeling Language (UML), a notation system used in object-oriented analysis and design.

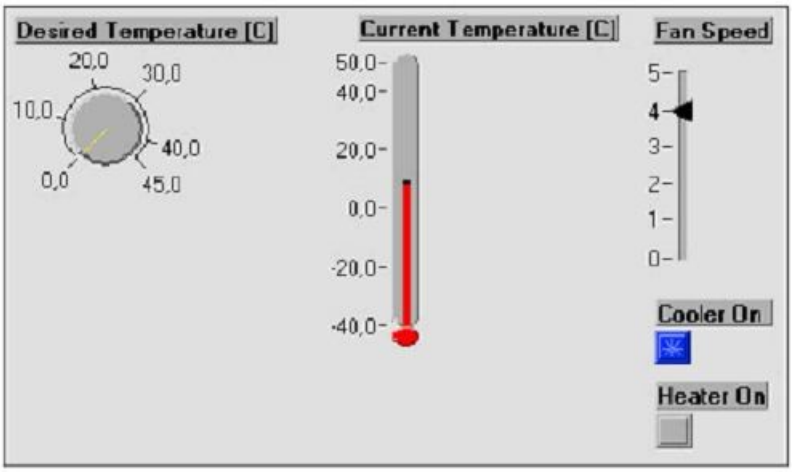
Subsystem

The Climate System component is comprised of five classes:

Climate Controller
Fan
Heater
Cooler
Temp Sensor

Design of Graphical User Interface

The picture beside represents the GUI. A knob in the upper left corner controls the desired temperature. On the right side, current system information is displayed.



11 Embedded Operating systems & Multitasking

11.1 Embedded Operating Systems

Some embedded systems have to deal with large amounts of data. They have to receive input data and they also need to output their results. In the beginning, users wrote their own computer programs to do this. Increasingly, many of these programs had similar features. After all, most of the devices operated the same way.

Also, with increasing computing power, more data was collected to be processed. Users had to organize their data on the hard disk. They also had to be able to transfer data between one another. They would also want automated ways of letting their programs run. Let's look at these tasks:

Getting input from a user
Outputting to a display and/or a printer
Creating, reading and writing to files
Load, run and terminate a computer program

Over the years, programs to do these tasks were collected and standardized so they could work with each other. Such a collection of programs was called an Operating System (OS). Common examples are Linux and the Windows family.

Smaller embedded systems may not need all the features of an operating system. In fact, these features, if included without thought, will increase the size of the embedded system needlessly and may even prevent the system from running! That is why there is a wide variety of embedded operating systems.

Let us look at the development of some of these concepts:

User Interface:
From text-only outputs of early computers, we now have GUI (Graphical User Interfaces).

File System:
This allows for the orderly management of data. Facilities may range from a simple 'copy' or 'move' to automatic recovery of data when a power failure occurs.

Task Management: Loading/Running/Terminating a program
Users can schedule a computer to load various programs to perform the tasks required of it. The computer has to keep track of every process that it is executing. It needs to reserve memory to run, hard disk space and to reclaim these resources when done.

Note that a computer can only execute one program at a time. But because it executes instructions at a much higher rate than humans, it can execute several programs a portion at a time, it can give the impression that it is doing several things at the same time. For example, a computer can do printing while allowing a user to enter data into a file, at the same time. This

feature is called multitasking and with the increasing power of embedded processors, is frequently being used.

An *embedded operating system* is an operating system operating for embedded

computer systems. These operating systems are designed to be very compact and efficient, forsaking many functions that non-embedded computer operating systems provide, and which may not be used by the specialized applications they run. They are frequently also *real-time operating systems*, requiring a fixed time to respond to I/O requests, Multitasking is one the main features of a real-time operating system.

11.2 Multitasking Systems

There are various definitions of the terms task, process and thread. In this chapter, we shall use a descriptive approach. A program as it exists on the disk contains both code and data. When loaded into memory by the OS, the code becomes a set of instructions in memory which is called a task. Tasks that are *separate* executing programs are called processes while tasks that are executed in the context of a *single* program are called threads. When we use the word tasks here, we are referring to both processes and threads. In embedded application, multi-threading is more commonly being used than multi-processing.

Multitasking is a method by which multiple tasks, also known as **process** and **threads** share common processing resources such as a CPU. In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by **scheduling** which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called **a context switch**.
When context switches occur frequently enough the illusion of parallelism is achieved.

Using multitasking operating system has two important benefits. First, the embedded system can perform more work in the same amount of time. This is due to the fact that operating system can avoid the unnecessary delays encountered in executing a task. For example, the OS has to run two tasks, Task A and Task B. Task A runs to a point in its execution where it must stop and wait for input (perhaps waiting for data to arrive from network port). In a single-tasking system, the system waits to complete Task A before going to Task B. In a multitasking system, there is no need to wait for Task A to complete its task. The OS can run Task B while Task A is waiting for input. When Task A received its input, the OS stops running Task B and returns to Task A. The second main benefit is the system has the capability to handle multiple tasks “simultaneously”.

11.3 Multiprocessing and multithreading

A program consists of algorithms expressed in the program code. A process is the activity of performing work according to these algorithms. As shown in Figure 11.1, a process consists of a collection of memory areas allocated to the process by the OS, plus the current CPU state. When a program is launched, the OS creates a process and allocates a collection of memory areas (code, stack, data, heap and system memory).

A process owns its memory area. A running process is also defined by the current

CPU state. The CPU state consists of the general processor registers (IP, SP, Flags and other CPU registers) that can be modified by the application program. When a process is not running, its CPU state is saved in the process's system memory area.

Running more than one process at the same time is called multiprocessing. An OS runs multiple processes by constantly switching from one process's context to another (context switching). A process context consists of its CPU state and memory areas. The OS switches context by suspending the execution of the current process, saving its CPU state into its system memory area, and loading the context of the next process to run. A scheduler program in the OS determines when to terminate the current process and what is the next process to execute in a multiple process environment. In a real-time embedded system, each task must be executed in a predictable manner and within the time constraints. The scheduler program must be able to meet these requirements to support a real-time operating system.

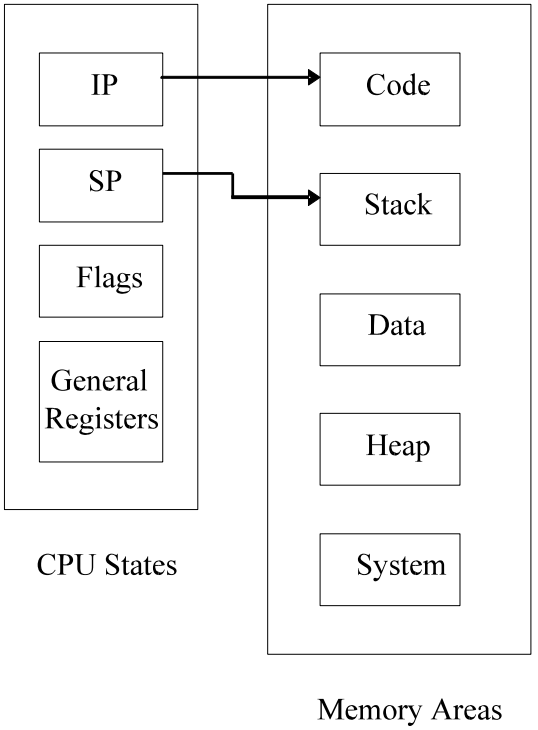


Fig 11.1 Context of a process

A thread is an independent flow of execution in a process. In a multithreading OS, a process consists of one or more threads. All threads in a process share the code, data, heap and system memory areas. As shown in Figure 11.2, each thread has a separate CPU state and a separate stack, a block of memory allocated out of the process' stack memory area. As all threads of a process share the same data and heap memory areas, all global variables in the process can be accessed by any of the threads. However, each thread has its own stack, all local variables and function arguments are private to the specific thread.

Because threads shared the same code and global data, they are tied much more closely than processes, and they tend to interact much more than separate processes. For this reason, *synchronization* objects are likely to be used more frequently in multithreading applications than multiprocessing applications.

Context switching between threads in the same process involves simply saving the CPU state for the current thread and loading the CPU state of the new thread. Because less

work is required for context switching between threads than between processes, threads are sometimes called lightweight processes.

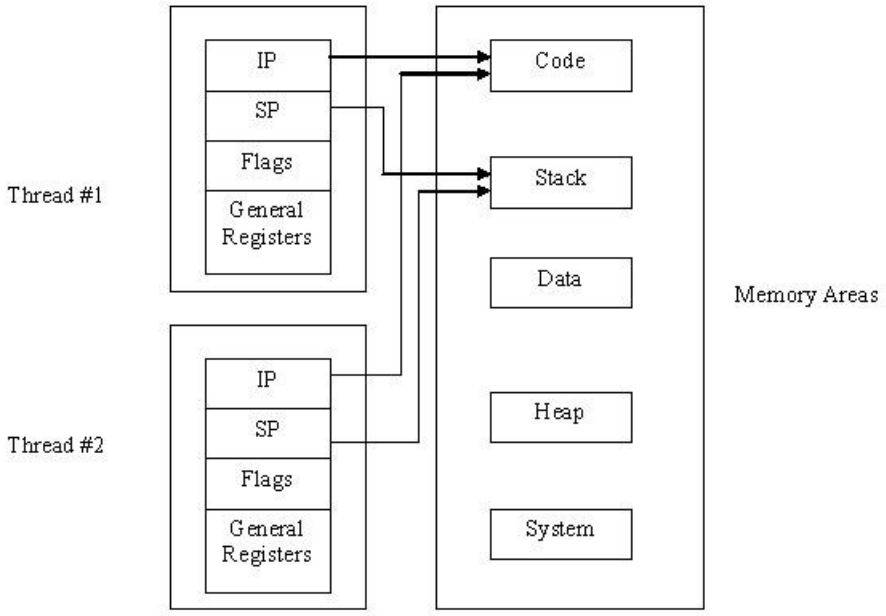


Fig 11.2 Context of a thread

Processes are normally created when programs are launched, either by user of the system or by another process calling the OS. In some systems, (such as Linux) processes can also *clone* themselves to create a new process, for example the fork() function calls. Processes created in this way often share the same code memory to conserve memory use, but they each get their own stack, data, heap and system data areas. Processes created by cloning may sound like threads, because they are running from the same program code. However, they are a process because they each have their own separate global data, unlike threads.

In summary, we see that threads are more desirable than tasks, as they use up less resource. However, threads have to be initiated by the programmer, whereas tasks are managed by the operating system as soon as a program is loaded into memory.

11.4 Scheduling

A multitasking OS worked by running a task and then switched to another task. This can be done in two ways:

i) cooperative - programs give way to one another when doing input/output for example, waiting for a user to press a key. This is very slow, from a computer point of view.
ii) Pre-emptive - programs forced to give way to one another, to prevent "hogging".

The amount of time an OS let a task execute is called a time slice, is often of a fixed duration. The OS partitions out a time slice for each task, one after another. When the time slice for

running a task ends, a program in the OS called the scheduler determines which task will have the next time slice.

A task can be one of the three states:

Running. In this state, the task is executing

Ready. In this state, the task is waiting for their turn for the CPU
Blocked. In this state, the task is waiting for something to happen.

A scheduler maintains one or more internal lists for keeping track of the state of each task. Typically, it has one ready list and a separate blocked list for each *synchronization object* on which tasks are waiting. The task at the head of the ready list is the next task to run. Tasks on any of the block lists are suspended. They are waiting for some events. Whenever an event occurs for which a task on the blocked list is waiting, the task is removed from the blocked list and placed onto the ready list, where it waits for its turn for execution.

How does a scheduler determine which task to run next? The answer depends on the scheduling algorithm used. Many type of scheduling algorithm are available, such as first in first out (FIFO) and round robin. For a real time embedded system, it is critical that the scheduling algorithm be deterministic, i.e. it is always possible to predict which task will run next. We will present the round robin scheduling here because it is simple and predictable.

Figure 11.3 shows how the round robin scheduling works. The contexts for the tasks that are ready to run are kept on a ready list, which is implemented as a linked list of task control blocks (TCB). Assuming all tasks has equal priority; the scheduler removes the TCB from the head of the ready list and makes its associated task the current executing task. This is referred to as switching-in the task's context. The task runs for a time slice, at the end of which the scheduler (awakened by the timer interrupt) saves the state of the task (referred to as switching-out the task's context), place its TCB at the end of list, pull the next TCB off the head of the list, and run the task. The scheduler continues to repeat the process, with each task running for an equal time slice, by proceeding through the list.

We have assumed the tasks have equal priority, but more often than not, some tasks are more important than others. The programmer can assign each task in the system a priority, a numeric value that assigns a level of importance to the task. When it's time for the scheduler to choose the next task to run, it selects the task from the ready list that has the highest priority. Implementing task priorities are important in real time scheduling algorithm. A real time OS has to implement deterministic scheduling, so the programmer can, given any circumstances, always identify which task in the application will run. The scheduler in a real time OS always selects the highest priority task on the ready list. It is immediately scheduled again and get the next time slice. By contrast, the Windows OS occasionally boosts the priorities of low priority tasks to ensure they get some CPU time. To guarantee that the system meets its real time deadlines, the programmer has to have complete control over which task or tasks are eligible to run.

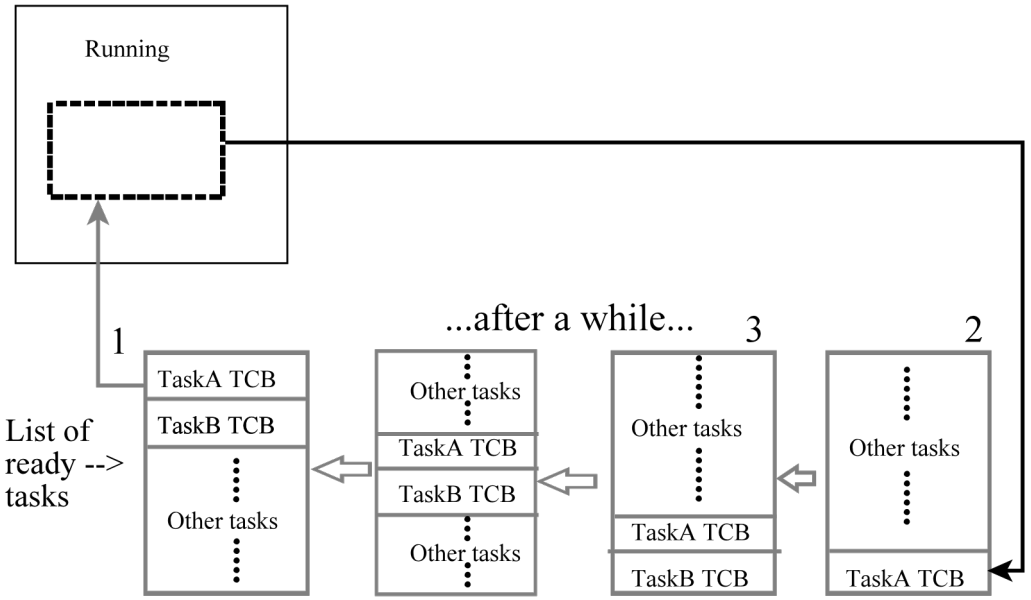


Fig 11.3 Round robin scheduling algorithm

11.5 Synchronization

In a multitasking system, tasks can interact. Sometimes they interact directly with each other; other times they interact through shared resources. These interactions must be coordinated, or synchronized, to prevent what is called a race condition. A race condition occurs when the outcome of the computation of two or more tasks depends on how quickly the tasks execute.

Let's have a look at what happens to make a race condition. Suppose both Task 1 and Task 2 access a global counter variable. Each task increments the counter variable using the following code:

```
Counter++;
```

After compilation, the generated processor might look like this (using pseudocode)"

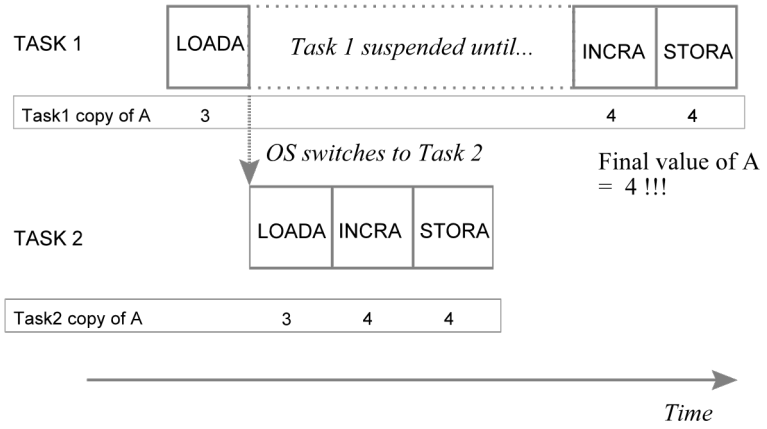
```
LOADA     counter
INCRA
STOREA    counter
```

The first instruction loads the value of counter into the CPU's accumulator. The next instruction adds 1. The final instruction stores the result back to counter.

Suppose Task 1 reaches the series of instructions when the counter has a value of 3. It executes LOADA instruction and load 3 into the accumulator. Before Task 1 can execute the INCRA instruction, however, the task's time slice ends and the scheduler switches it out. The content of the accumulator (a value of 3) is stored with the task context when it is switched out.

Task 1 resumes

The scheduler starts Task 2. Task 2 reaches these instructions and executes LOADA. Because Task 1 did not store the result in counter, Task 2 fetches 3 into the accumulator also. Task 2 then executes the INCRA and STOREA instructions and continues with whatever instructions follow. The content of global variable counter is now 4.



At some time later, Task 1 is rescheduled. The scheduler reloads Task 1's context and places 3 back to the accumulator. You can now see the problem. Task 1 restarts, executes INCRA and STOREA instructions and proceed with whatever instructions follow. The content of the global variable counter is again 4. Had Task 1 not been preempted at the moment it was, it would have executed the INCRA and STOREA instructions, and the global counter would be 5. Because neither task guarded its access to the counter global variable, a race condition occurred.

The code section that accesses the shared resource is called a critical section. In the example above, the critical section is just one C statement. To avoid race conditions, we need tasks to be *mutually exclusive*, to ensuring that only one task can be executing in a critical section at any moment.

Synchronization objects are used to synchronize the use of shared resource and prevent race conditions. We will look at three types of synchronization object,

Mutex objects
Semaphore objects
Event objects

11.5.1 Mutex Objects

A mutex object, sometimes called a critical section object, derives its name from its use in coordinating mutually exclusive accesses to a shared resource. A mutex object can be in one of the two states, owned or free. The mutex object can be owned by only one task at a time. Operating systems that support mutex provide two function calls for manipulating them: a wait call and a release call.

Operating systems that support mutexes provide two function calls for manipulating them, a wait call and a release call. When a task wants to acquire a mutex, it issues a wait call. If the mutex is free, the wait call returns immediately and the calling task has acquired ownership of the mutex. If the mutex is already owned, the wait call doesn't return, and the calling task is blocked. A task waiting on the mutex becomes unblocked when the owner of the mutex issues a release call on the mutex.

A mutex is similar to a room built around a critical section of code. The room has one door in and one door out. Furthermore, a guard at the door admits only one person (task) into the room (critical section) at any given time. Tasks arriving at the entrance while the room is

occupied must wait at the door until the room becomes empty. Also, the guard (operating system) sees to it that those waiting at the door are admitted in task priority order.

11.5.2 Semaphore Objects

A semaphore consists of a data item, which we will call a count, and a pair of operations, *wait* and *release*, which are implemented as OS APIs. A task requesting access to whatever resource the semaphore guards need to perform a *wait* operation on the semaphore. When a task is finished with the resource, it performs a *release* on the semaphore.

The semaphore's count determines what happens when a wait or release operation executes. When the program creates a semaphore, it sets the semaphore's count to an initial positive value. The value indicates the number of tasks the semaphore will let pass before closing the door. Whenever a task performs a wait operation on the semaphore, the OS first checks to see whether the semaphore's count is greater than 0. If so, the call operation succeeds (task is allowed to continue) and the semaphore's count is decrease by one. If the count is 0, the task is blocked.

When a task has finished using whatever resource the semaphore is guarding, the task issues a release call on the semaphore. The release call causes the OS to increment the semaphore's count by 1. Semaphore objects allow multiple tasks to simultaneously access a common resource, such as network connection.

11.5.3 Event Objects

Event objects are typically used for synchronizing tasks processing rather than for controlling access to shared resources. It is a mechanism that lets a task go to sleep until, for example, some data is ready for it to process or a request is ready for it to service.

Operating system functions allow tasks to set to **signaled** as well as reset to **nonsignaled** an event object. Event objects can be created in either the signaled or the nonsignaled state. When a task waits on a non-signaled event object, the task is blocked until another thread sets the event object. When a task waits on a signaled event object, the task does not block. A common use of an event object is to notify a task when a pending I/O operation has been completed.

There are two types of event objects, i.e. auto reset events and manual reset events. Auto reset events are automatically reset to non-signaled state when a single waiting task is released (unblock). If multiple tasks are waiting, only the highest-priority waiting task is released. The next time a running task set the event, another waiting task is released, and so on. Manual reset events remain set to the signaled state after a task sets them. If multiple tasks are waiting on the event, they are all released (unblocked) when another task set the event. Subsequent tasks that wait on the even object proceed immediately, without blocking. To cause task to block on the event again, a task must explicitly reset the event to its nonsignalled state. An example of setting up multiple threads is shown in the appendix.

Appendix

Creating multithreading application

When a process begins, it contains a single thread. Additional threads within the process must be started explicitly. A thread can be started by a call to *pthread_create()*,

The *pthread_create()* function's prototype is

```
int pthread_create(pthread_t *thread, const phread_attr_t *attr, void *(*start_routine)(void *), void *arg)
```

Parameters

thread - Once the function is called successfully, the variable whose address is passed as first argument will hold the thread ID of the newly created thread.

attr - contain certain attributes which we want the new thread to contain. It could be priority, stack address, stack size etc. Set to NULL if default thread attributes are used.

*void * (*start_routine)* - pointer to the function to be threaded. Function has a single argument: pointer to void. Each thread starts with a function and that functions address is passed here as the third argument so that the kernel knows which function to start the thread from.

arg - As the function (whose address is passed in the third argument above) may accept some arguments, we can pass these arguments in form of a pointer to a void type. A void type was chosen because if a function accepts more than one argument then this pointer could be a pointer to a structure that may contain these arguments.

Initializing a Mutex

The *pthread_mutex_init()* function initialises the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
```

Parameters

mutex - the address of the variable to contain a mutex object.

attr - the address of the variable containing the mutex attributes object..

Locking and unlocking a Mutex

After initilizing a mutex, any critical region in the code can be locked using the *pthread_mutex_lock()* function.

The *pthread_mutex_lock()* function's prototype is

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

Parameters

mutex - the address of the mutex to lock

If the mutex is already locked by another thread, the thread waits for the mutex to become available. The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it.

To unlock and release the mutex, the function to call is *pthread_mutex_unlock()*.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Waiting for thread termination

pthread_join() is used when one thread waits for another one to finish. The two threads work in parallel, then they must combine the results they obtained. One of them calls join and waits for the other one to exit, so it can collect its result.

```
int pthread_join(pthread_t thread, void **status)
```

Parameters

thread - the thread to wait for

status - the location where the exit status of the joined thread is stored. This can be set to NULL if the exit status is not required.

Wait on a condition

To let a thread sleep, condition variable can be used. In C under Linux, there is a function pthread_cond_wait() to wait or sleep.
On the other hand, there is a function pthread_cond_signal() to wake up sleeping or waiting thread.
Threads can wait on a condition variable.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
```

Parameters

cond – pointer to the condition variable.

mutex - Pointer to the mutex associated with the condition variable *cond*

12 Embedded C

12.1 Introduction

C is a high-level language, developed in the 1970's at Bell Labs. Used in developing the Unix operating system, it was optimized for system development. Unix was made readily available for the academic community. Along with that, C came along and became very popular. This original version is popularly known as the "K&R" version, after its authors, Brian Ritchie and Kernighan. As various vendors implemented the language, they added in vendor specific extensions, to correct deficiencies in the language. This had the effect of losing the advantages of portability. The American National Standards Institute (ANSI) formed a committee to standardize the language and include in many new features. This was finalized in 1989 and this was known as ANSI C. From that time, the standardization task took on a global nature under the International Standards Organization and this was known as Standard C. Much of this effort involves providing support for multinational applications. Of course, there is now C++ and Java, later developments.

12.1.1 Advantages

Portability: same program source can run across variety of processors - 4/8/16/32 bit with little modifications. Quick upgrade without much s/w change!

Efficient: program code can be as small or fast as an assembler program

Productivity: As in all high-level languages, details of memory spaces, hardware resources are taken care of by compiler. For example, there is no need to worry about location of data, whether ROM or RAM.
Concentrate on problem, not on machine characteristics!
May be possible to write program on PC, test using PC compilers, then bring over to target processor.

12.1.2 Disadvantages

cryptic: program can be hard to read (often called a write-only language!) many non-intuitive operators and their order of evaluation can cause confusion

protection: indiscriminate use and management of pointers can corrupt system very easily

debug: difficult to on development tools that use assembler only

12.2 Using C on an embedded system

Various standards groups like ANSI have specified how the C language is to be used. These specifications are oriented towards desktop computers, workstations and larger systems. In these systems, processor storage consists of RAM for the operating system and user applications. This is available in terms of several megabytes. Secondary storage runs into the thousands of megabytes. The processor runs at several hundred Million of Instructions per Second (MIPs).

12.2.1 Cross compilers

Modern C compilers which run on desktop machines, but which produce code for another processor are called cross compilers. Cross compilers for embedded systems provide various modifications to standard C called extensions This helps in using the processor more efficiently. They also provide various header and include files to access processor resources more easily, like predefined register names.

This may also make the program less portable. But the objective is to use the lowest cost processor for the project. However, the programming job may overall be more efficient than using assembler.

12.2.2 Design constraints for embedded systems

A typical embedded system has much more modest resources. For lower end processors, a few kilobytes of non-volatile program memory, a few hundred bytes of RAM and a processor speed of up to 1 MIP. Of course, there are embedded systems which have as much computing power and resources as a desktop the only difference being the lack of a keyboard, screen and mouse.

In order to make best use of these resources, the programmer must take note of the machine architecture when writing programs. The embedded system typically has a few kilobytes of ROM for program, but more crucial, only a few hundred bytes of RAM on chip. If we exceed this, extra memory chips must be added to the hardware and possibly decoding hardware. This also increases the size of the printed circuit board. For a mass-produced device, this can drive up the cost significantly. The overall product may be bigger as well. The goal here is to use all the on-chip facilities as efficiently as possible.

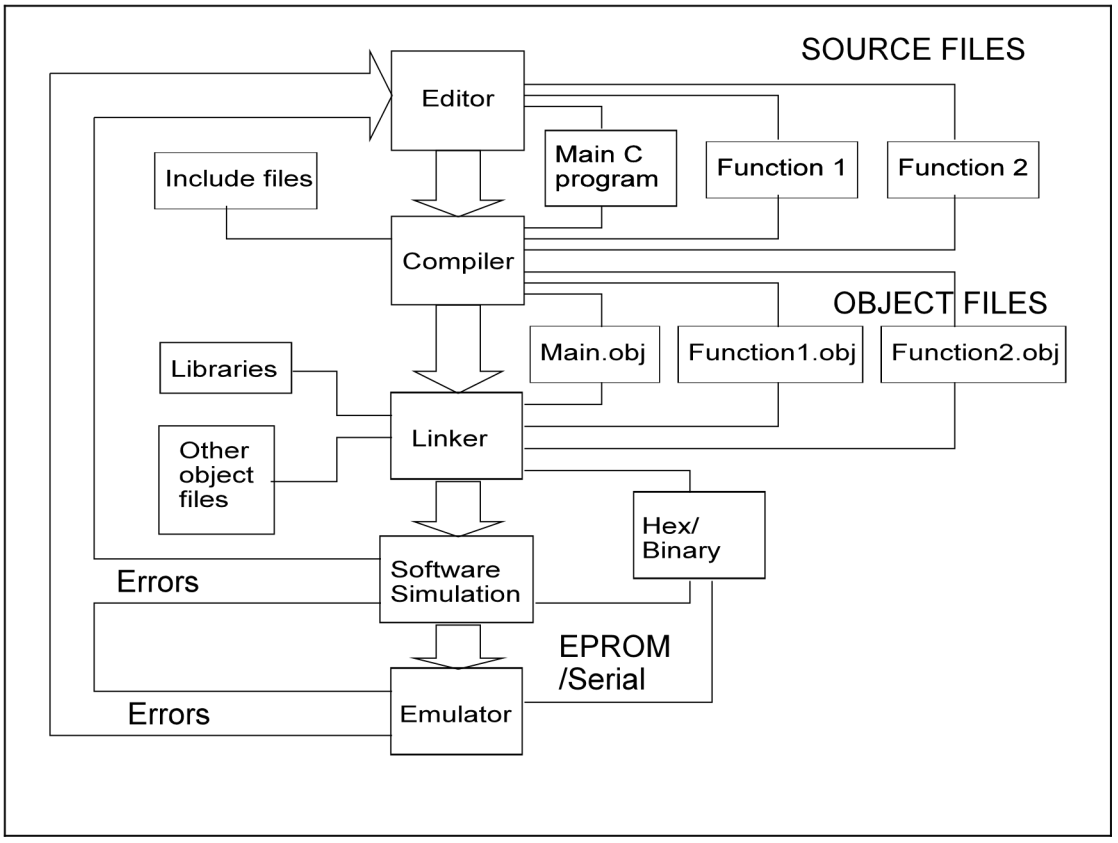
12.3 Development of an embedded C program

An embedded system is part of another product. For example, the processor in an air conditioning unit is part of, or “embedded” in the main air conditioning unit. The function of the embedded system is fixed, because of its embedded software.

There is more than one way to implement a software function. Because of this, we need to look at how the cross compiler generates assembly or machine code, in order to make better use of the processor. This has important consequences when debugging a system.

12.3.1 Differences in normal C and assembler programming

Writing an embedded C program is very similar to that of a typical C program. The main difference is the conversion and transferring of code to the system, which is through an EPROM or serial communications.



Cross Development Cycle of an Embedded C Program

When compared to standard assembler programming, the main difference here is the debugging method.

12.3.2 Debug facilities

Some considerations to take note of are the choice of debugging languages. It is preferable to debug in C. But the processor executes machine instructions. One line of C code may generate several bytes of machine code. It is a very useful feature to be able to debug in C, assembler, or a combination of them, in both the simulator and emulator.

The simulator and emulator however, work mainly at machine level. These systems need to know the variable names in the source file and where they are in the processor memory. Also they need to know which C source line corresponds to which machine code it is executing. All this information are produced by the compiler and has to be transferred to the software used for debugging, like the simulator emulator. The most common practice is to include this information *into* the executable so that the debug version of the executable will be larger in size than the production version. An example is the use of the open sourced gdb (GNU debugger) system which has been ported to many systems, including Windows and Linux.

12.4 Some features of using embedded C

For general purpose desktop microprocessor system such as PC's, all program and data functions are in RAM. But for microcontrollers we have to be concerned about:

CODE (Program) is normally in EPROM - read only, not writable
DATA (Variables) is normally in RAM - read/write, volatile limited RAM space
C function and their use of stack space for local variables

Since a microcontroller has to control devices, there are several features in this type of programming that differs from normal C programming. For example, accessing processor registers that control the timer or interrupts. These registers have standard names and it is convenient to use them. Also the standard input/output devices which are taken for granted like the graphics display, keyboard, mouse and hard disk, are not present. Later we will look at ways of optimizing C programs.

12.4.1 Accessing processor resources

The definitions for the Input/output registers are normally kept in header files, and should be included in your C program. Thus it is possible to refer to these registers by just referring to them appropriately.

Standard library functions for desktop machines like printf, scanf and so on, may not be available. Input and output may be to serial devices. It is common for compiler vendors who provide libraries based on this fact. For other ports and other kinds of facilities like string routines, multiple precision arithmetic and floating point calculations, users have to write their own routines. It is important to see what features are available and not make assumptions.

The C compiler generates code that is memory location independent. However when linking, you have to specify where the code (ROM) and data (RAM) memory locations are. This is highly dependent on the system design. Also note that the generated code is not memory location independent.

Uninitialized variables are not guaranteed to be zero. You need to do it explicitly, or modify the C start up routine.

12.4.2 Accessing I/O devices

Type casting for memory mapped I/O

Since the task of a microcontroller is to control devices, interfacing plays an important role. For I/O devices which are addressed as normal memory, (this is called memory mapped) we treat as external memory and a normal read from an address is:

```
data = *((int *) 0x1000);      /* 0x1000 is address $1000 */
```

However this is a 16 bit operation - we have to make the result fit into 8 bits. This is done through a "cast" which is the purpose of (char *).

to read: `data = *((char *) 0x1000);` to write:

```
*((char  *) 0x1000) = data;
```

I/O mapped I/O

Since many embedded systems use I/O mapped operations, extensions to the C language allow the use of input and output statements.

to read: `data = _inp(0x332);`

to write: `retcode = _outp(0x331, data); /* retcode for status of op */`

Optimising compilers

Another consideration are optimising compilers. These try to make the user program smaller or faster. They do this by examining the program to remove what it considers redundant code. For example, a common situation with external devices is that values present in their registers change without the processor taking any action. For example, in a keypad - the value read from it changes depending on whether someone has pressed a key, without intervention from the processor.

As discussed before, we will output a column scan value and then immediately read back to see if someone has pressed a key.

```
#define KbdPort  *((char *)0xA000)/* ptr to keypad port */

KbdPort = Col7Lo;                /* (1) output a value  */
ScanCode = KbdPort;              /* (2) read is ignored */
```

Here the value retrieved into ScanCode comes from the keypad port.

If this is compiled it will not work. The compiler's optimiser assumed that, because no WRITE occurred between (1) and (2), KbdPort cannot have changed. Hence the code

generated to make the second access to the keypad port is optimised out and we will get ScanCode being equal to Col7Lo. This is shown in the equivalent assembler code:

```
KbdPort = Col7Lo;
     ldab 0FF stab
     0A000
ScanCode = KbdPort; stab
     _ScanCode
```

The solution is declare KbdPort as "volatile" thus:

```
#define KbdPort  *((volatile char *)0xA000)
```

Now the optimiser will not try to remove subsequent accesses to the register, as shown by the “ldab”.

```
KbdPort = Col7Lo;
     ldab 0FF stab
     0A000
ScanCode   =   KbdPort;
     ldab   0A000  stab
     _ScanCode
```

12.5 Generating efficient code

As mentioned earlier, because of the limited resources available to an embedded system, we need to look at various considerations to make code more efficient in terms of speed and code size.

These typically depend on the word size of the processor - be it 8, 16 or 32 bits. Accessing and processing variables of longer length than this will generate more instructions and use up more memory. In this section, we show some examples of the code generated by a typical compiler. You will see that the choice of variables will affect the code generated.

12.5.1 Use short and/or unsigned variables

The objective is to conserve use of scarce internal RAM. Try to use 'unsigned char' types for unsigned quantities or ‘short int’ as they are one byte. Note that 'int' (integer) is 16 bit. For a one-line statement involving addition, the size of code produced is shown. Variables with s,i,u below are short (8 bit signed), integer (16 bit signed), unsigned integers (8 bit unsigned), respectively. Mixed variables use the most code as they have to convert from one form to another.

short	int	mixed	mixed unsigned
`sl=s2+ s1*s3;`	`i1=i2+ i1*i3;`	`i1=s2+ s1*i3;`	`ui=us2+ us1*ui3;`
9 bytes	12 bytes	24 bytes	18 bytes

12.5.2 Use bit flags

Use bit flags for variables that take on values of only true or false, or 1 and 0. Thus instead of using eight bytes for eight variables, you only need a byte.

There are six operations that can be performed and they work on a bit by bit basis, in the same way as their assembler equivalents. The following summarizes the operation of bit variables. The variable A here has the value 0x55 or %01010101 binary. In the 6811 equivalent, assume accumulator A contains the value.

Operator	**C**	**Result**	**Binary**	**Comment**
AND	`A & 0x0F`	0x0A	0101 0101 0000 1111 --------- 0000 0101	Bitwise AND
OR	`A | 0x0F`	0x5F	0101 0101 0000 1111 --------- 0101 1111	Bitwise OR
XOR	`A ^ 0x0f`	0x5A	0101 0101 0000 1111 --------- 01011010	Bitwise XOR
NOT	~A	0xAA	0101 0101 --------- 1010 1010	One's complement
Left Shift	`A << 1`	0xAA	0101 0101 --------- 1010 1010	shift multiple, ignore CY
Right Shift	`A >> 1`	0x2A	0101 0101 --------- 0010 1010	shift multiple, ignore CY

Bit flags can be set up in one of two ways:

1. Use defines

```
#define STATIC 01
#define EXTERNAL 02
unsigned short flags1;
```

2. Use a structure

```
struct { unsigned short static:1;
   unsigned short external:1;
   }flags2;
```

To set a bit flag, look at the following examples:

```
   flags1 |= (EXTERNAL | STATIC);
   flags2.external = 1; flags2.static =
   1;
```

Note that in the structure, the most significant bits are assigned first. To branch on bit flag values, here are some other examples

```
if((flags1 & (EXTERNAL | STATIC))==0) s++; if((flags2.extenal==0)
&& (flags2.static==0)) s++;
```

12.5.3 Use ROM space for variables

Tables, messages, and other non volatile data should not be kept in RAM. Typically, they use up a lot of space and should be kept in ROM always. Be careful to see that they do not get copied to RAM by the compiler's code. Use the 'const' directive to store in EPROM. For example:

```
const unsigned char ScanTable [12] =
{0x7D, 0xEB, 0xED, 0xEE, 0xDB, 0xDD, 0xDE, 0xBB, 0xBD,
0xBE, 0x7B, 0x7E};
```

12.5.4 Arrays

If an array element or set of elements is frequently used, then efficiency will be increased by setting a pointer to the element and using the pointer to refer to it. If possible, use one dimensional arrays. Higher orders will involve multiplication, which is always slow.

```
example()
{ short arr[20];
   short I,*p;

   t = arr[ I ]; p
   = arr + I; t =
   *p; p = arr + I;
   t = *p;
```

12.5.5 Functions

If function is non-recursive (does not call itself) and does not require local (automatic) variables, then the stack allocation / deallocation overhead on function entry and exit can be avoided by using:

```
function ( args )
/* argument declarations here */ {
       /* a non-recursive function with no local variables */
}
```

For functions, try not to pass data in functions:

```
function (argument1, argument2, argument3);
```

This takes up space on the stack

12.5.6 Use global variables

Function parameters take up stack space. Also, local variables in a function take up extra RAM. One way to save on this is to use global variables. Note that this is not recommended in C, because modules may modify a global variable by accident. But they may save RAM space and improve speed.

```
       unsigned char       argument1, argument2

              function() result1 =
              argument1 ... result2 =
              argument2 ...
```

12.5.7 Interfacing to assembler

As we have seen in an earlier section, one C statement can generate many bytes of assembly code which takes time to execute. If certain parts of a program need to be speeded up, we may need to write that part of the program in assembler.

Of course this assumes one is able to come up with a more efficient algorithm to achieve the task! Therefore it is useful to examine the code produced by a compiler, to see if the compiler has done a good job of generating the machine code. We can interface to the main program using standard linking techniques and inline code.

12.5.7.1 Linking object modules

Most cross compilers allow you to assemble program modules separately and combine them at link time.

12.5.7.2 Inline code

This inserts actual assembler instructions into the C program. The compiler needs to generate assembly language code from the C program, then invoke the assembler to convert the intermediate program into the object file.

Use asm ("CLI") directive to embed an assembler portion

If the assembler program needs to access C program variables, you need to be careful of what its 'real' name actually is:

EG: variable is 'ScanCode' - internally it becomes '_ScanCode'

Advantages
You do not have to maintain a separate file of assembler routines. Everything is in the C program.

If we are linking in an external assembler routine, it is not possible to know the absolute locations of program code and variables when debugging. Using inline code, this is possible as all addresses refer to the same source program.

Disadvantage
As the compiler has to generate an intermediate assembler file, it is slower. Also the assembler has to be called up after that.

12.6 Other Considerations

Macros and functions. For faster execution, use macros; to minimize memory, use functions.

Singapore Polytechnic
School of Electrical and Electronics Engineering
ET0104 Embedded Computer Systems

Tutorial 1 Introduction to Embedded Systems

1. For a computer system indicate which bus or buses are being described.

   a) A unidirectional bus.
   b) Carries signals used to synchronize data transfer operations.
   c) The CPU uses this bus to select a specific memory location for data transfer.
   d) During a WRITE operation, this bus carries data from the CPU.
   e) The number of lines on this bus determines the maximum memory capacity.
   f) The number of lines on this bus determines the memory word size.

2. (a) An embedded system does not use Graphical User Interface. Therefore they should not need to use powerful processors. Explain if this statement is true.

   (b) In what way does increased processor power in desktops benefit embedded systems?

   (c) Compare between the Intel line of processors and those based on the ARM processor.

   (d) What are some ways computers increase in performance.

3. Describe some differences in characteristics between desktop and embedded systems.

4. i) What are three general areas to consider in power saving design for embedded systems?
   ii) In the area of power supply design, what are three factors to consider?

5. In many battery powered electronic projects, it is common to use a 9V battery connected to a 7805 regulator. A typical project consumes 50 mA of current: what is the efficiency of the power supply? If now a switching power supply with 85% efficiency and a LDO regulator that needs only 0.5V between input and output voltage is used, what is the overall efficiency?

6. Compare and contrast the differences between using software loops and hardware for timing functions.

Singapore Polytechnic
School of Electrical and Electronics Engineering
ET0104 Embedded Computer Systems

Tutorial 2 Bus Systems and Devices

1. a) What are the address and data bus sizes (memory and I/O) of the 8080 mode bus?
b) Does the bus differentiate between I/O and memory operations? How does it do this.
c) Compare and contrast the two approaches to I/O addressing.
d) What are some devices that use the 8080 mode bus?

2. Modern digital cameras have computer embedded systems built in. They need to be able to take photographs *quickly*, store and delete them as necessary. The user preferences have to be stored as well.

i) Write down the types of memory you would use in the design.
ii) For each type of memory, describe what you would use it for.
iii) Explain your choice as well, especially for flash types of memory.

Think of general camera use, and do not go into too much detail.

3. Compare and contrast between a BIOS and a monitor program.

4. In the design of input/output ports, you can use either buffers or latches. Discuss which type you would choose if you wanted to build:

(a) An 8-bit input port (b) an 8-bit output port

5. Describe the boot up process for a PC. How does the BIOS help in the boot up process?

Singapore Polytechnic
School of Electrical and Electronics Engineering
ET0104 Embedded Computer Systems DECC 3FT/4EO

Tutorial 3 Address Decoding with I/O Devices

1. What is memory foldover and how can it be eliminated?

2. The system BIOS resides at the *highest* 16K of a microprocessor using an 8080 mode addressing scheme where the address bus size is 16 bits. Using 4K memory devices, design a decoding scheme. In your solution: (hint: What is the highest 16 bit address). Design a decoding circuit using the 7485 and a 74138.

3. An I/O board using an 8080 mode bus needs 4 buffers and 4 latches. Its base address is 20H. Design a suitable decoding scheme for this board.

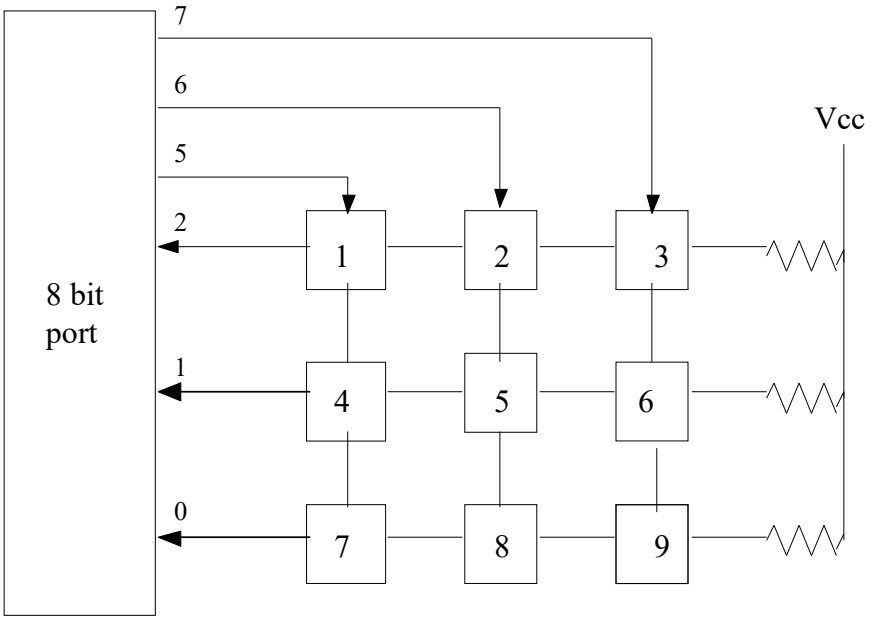
4. What is an advantage of using the 7485 in decoding?

5. Compare and contrast the differences between memory and I/O decoding on an 8080 mode bus.

6. An I/O board for a Secondary Memory Interface bus is to be interfaced to 14 switches and two 7 segment LEDs. How many buffers and latches are required?

Singapore Polytechnic
School of Electrical and Electronics Engineering
ET0104 Embedded Computer Systems

Tutorial 4 Keypad



1. For a 3x 3 matrix key arrangement, specify the word value for scanning each of the columns and fill in the blanks. If the key scanning method is same as the example given in the lectures, calculate the key scan values and fill in the blanks accordingly.

```
#define Col7Lo    ____          /*  column 7 scan */
#define Col6Lo    ____          /*  column 6 scan */
#define Col5Lo    ____          /*  column 5 scan */

const unsigned char ScanTable [12] =
/*    1    2    3   */

{    ----, ----, ----,
/*    4     5     6   */

     ----, ----, ----,
/*    7     8     9   */

     ----, ----, ----
};
```

---

2. For a *simple* key input application as drawn in the book, the keys are active low. Suppose the buffer address is 320H, the following instructions will be used to detect whether there are key(s) pressed (It does not matter which key or keys are pressed). Fill in the blanks.

```
KeyPress = ___________ ;

if (KeyPress != _______);

     ProcessKey();
```

3. What are some of the methods used to solve the problem faced when more than one key is pressed simultaneously on a keypad?

4. There is a requirement for a keyboard with 90 keys. Explain your choice of key layout.

Singapore Polytechnic
School of Electrical and Electronics Engineering
ET0104 Embedded Computer Systems

Tutorial 5 LCD

1. How do you adjust LCD's contrast (appearance of dark over light areas)?

Describe what is meant by DDRAM, CGRAM and CGROM. What are their functions in an LCD module.

What signals are needed to interface an Alphanumeric LCD module.

Briefly explain their functions.

2. What factors should be considered when selecting a LCD module?

3. In controlling an LCD, before sending the next instruction, we must ensure the previous operation already finished. What methods can be used for that purpose?

4. A 16 X 1 line LCD module is connected to a bus interface through an 8-bit interface. Figure out the control instruction values in tabular form including RS, R/$\overrightarrow{W}$, and data bus, to initialize the LCD and then display "Poly" on LCD, shifting the cursor to the right after writing a character (5x7 dots).

5. Do the same for the above question if the interface is 4 bit.

Singapore Polytechnic
School of Electrical and Electronics Engineering
ET0104 Embedded Computer Systems

Tutorial 6 Stepper Motor

1. Briefly describe the basic structure and characteristics of a Permanent Magnet stepper motor.

Describe and compare the possible methods for position control of stepper motor.

What are some of the issues concerned for speed control of stepper motor?

2. A stepper motor has a step size of 3.6 degrees. What should be the stepping pulse rate to drive it at a speed of 2 revolutions per second?

If the motor is to rotate at a rate of 12rpm, calculate the period of the stepping pulse.

3. A stepper motor with 5 phases (that is, it has 5 windings or magnetic poles labelled A, B, C, D, E) is controlled by a 8 bit output latch (bit0 controls phase A, bit1 for phase B, and so on).

Calculate the control values to be sent to the latch so as to make the motor rotate one cycle.

Show these values in binary form, for both full step and half step control of the motor. How do you send out these control values to make the motor rotate in reverse?

Singapore Polytechnic
School of Electrical and Electronics Engineering
ET0104 Embedded Computer Systems

Tutorial 7 DACs & ADCs

1. What are the main elements of a typical DAC device?

What is the relation between the DAC output and the input digital value?

2. A 8-bit DAC generates 4.5V output for digital input 0xFF. We want to generate a half-wave rectified sine waveform, with the peak voltage Vpeak=2V.

Suppose one cycle of the rectified waveform will use 6 points, calculate the corresponding digital values.

If the frequency of the output signal should be 300 Hz, what will be the time delay between sending two consecutive values to DAC?

3. The digital output of a 10-bit binary counter are used to supply the digital inputs to a 10 bit binary coded DAC.

a) Sketch the analog output waveform produced by the DAC if the counter is incremented up by a 40 KHz clock signal. What is the frequency of this output wave?

b) What happens if the counter counts down only, or up and then down?

4. An 8-bit DAC outputs a voltage of 4V for a digital input 0110 0100B.

a) Determine the output voltage for an input code of 1011 1001B.

b) How many bits are required for the DAC so that its full-scale output is 10V and resolution is less than 40mV?

5. What are main techniques used for A/D conversion?

Briefly describe their working principles and compare their advantages and disadvantages.

Singapore Polytechnic
School of Electrical and Electronics Engineering
ET0104 Embedded Computer Systems

Tutorial 8 Top Down Design

1. a) AA typical microprocessor may cost little. However products based on it may cost a great deal.@ Comment on this statement.

b) Why do we need to spend effort in obtaining product requirements?

c) Would the system requirements of a product remain the same throughout its design and development cycle? Explain your answer

2. You are to design a battery charger using a microprocessor, so it will have more features than standard ones. The user needs to select a battery type which will control the time and amount of current used. When charging, an LED will indicate this. When fully charged, the LED will change colour to show this. The battery type selected is shown, together with the status of charging.

a. Specify the goals and constraints
b. Identify the sub-systems of your system.
c. Draw a *use case* diagram for the above system.
d. Draw a sequential interaction diagram for all the *use case* listed.

3. a) What is system analysis? What areas must be considered when doing system analysis?

b) During the system analysis phase, why is it necessary to first identify the hardware subsystems?

c) What is the purpose of conducting feasibility studies during systems analysis? How is this different from a prototype?

4. What are some reasons for using a bought-in system to one that is designed from the beginning?

5. What are the factors that would influence your choice of microprocessor?

Tutorial 9 Graphics Display Technology

1. Describe how a digital image is stored in a computer.

2. i) What are the advantages and disadvantages of 8-bit images?
ii) How are they stored and displayed?

3. What are the types of display available? Give examples and discuss their merits / shortcomings.

4. What are the differences between passive matrix and active matrix display?

5. What are some of the interfaces available in GPIO pins that can be used to connect displays?

Singapore Polytechnic
School of Electrical and Electronics Engineering
ET0104 Embedded Computer Systems

Tutorial 10 GUI

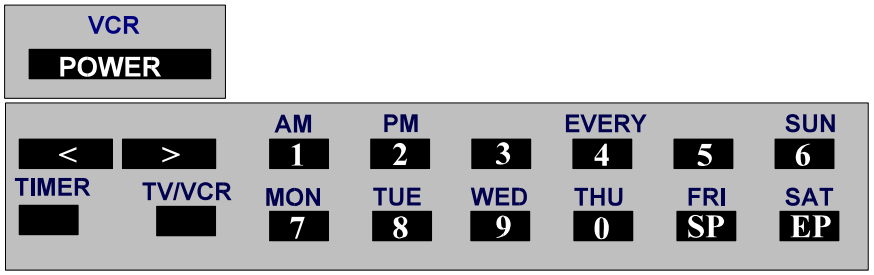
1. Discuss three principles to consider for a User Interface Design.

2. GUI are found in many common applications. Give three practical examples where GUI is applied.

3. Name three different types of GUI interaction, and give two examples for each application.

4. Give three guidelines that will promote the effectiveness of user-interface design. Give one reason for each case.

5. Comment on the following design for the control of a Video Recorder.



You should use the guidelines in the chapter on Graphical User Interface to assist you.

Singapore Polytechnic
School of Electrical and Electronics Engineering
ET0104 Embedded Computer Systems

Tutorial 11 Embedded Operating Systems and Multitasking

1. Discuss why an Operating System (OS) might be useful for an embedded system. Under what situations, one might not want to include an OS into the embedded system?

2. i) What are the differences between a process, task and thread?
ii) What are the three possible states of a task?

3. What are the differences between cooperative multitasking and pre-emptive multitasking?

4. i) Multitasking improves the efficiency of an embedded system. Discuss two points to support this statement.
ii) Discuss the functions of a scheduler and identify two types of scheduling algorithm used by a real-time operating system.

5. Why are there needs for synchronization in multitasking environment? How can synchronization be achieved in multitasking environment?

Singapore Polytechnic
School of Electrical and Electronics Engineering
ET0104 Embedded Computer Systems

Tutorial 12 Embedded C and Operating systems

1. What are the advantages of using C language in microcontroller programming?

What are some of the differences between normal C and embedded C programming?

What kinds of functions are better written using assembly language?

2. List some measures to make Embedded C program code more efficient in terms of speed, code size, and usage of microcontroller's resources.

3. What is the C code to input 8 bit data from I/O address 0x132, similarly how do you output 8 bit data to I/O port at address 0x134? Use your own variable names.

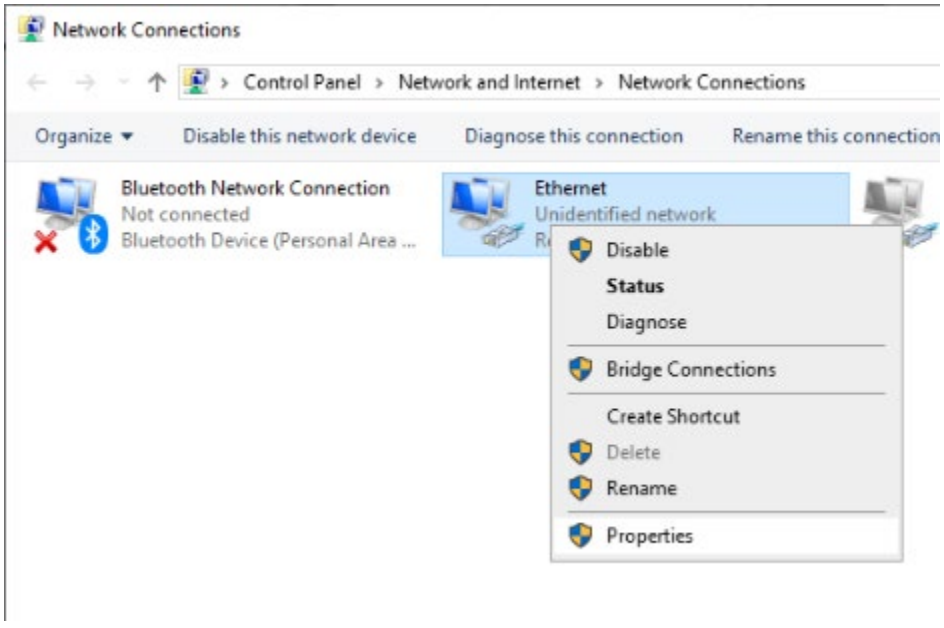
SINGAPORE POLYTECHNIC
SCHOOL OF ELECTRICAL AND ELECTRONICS ENGINEERING

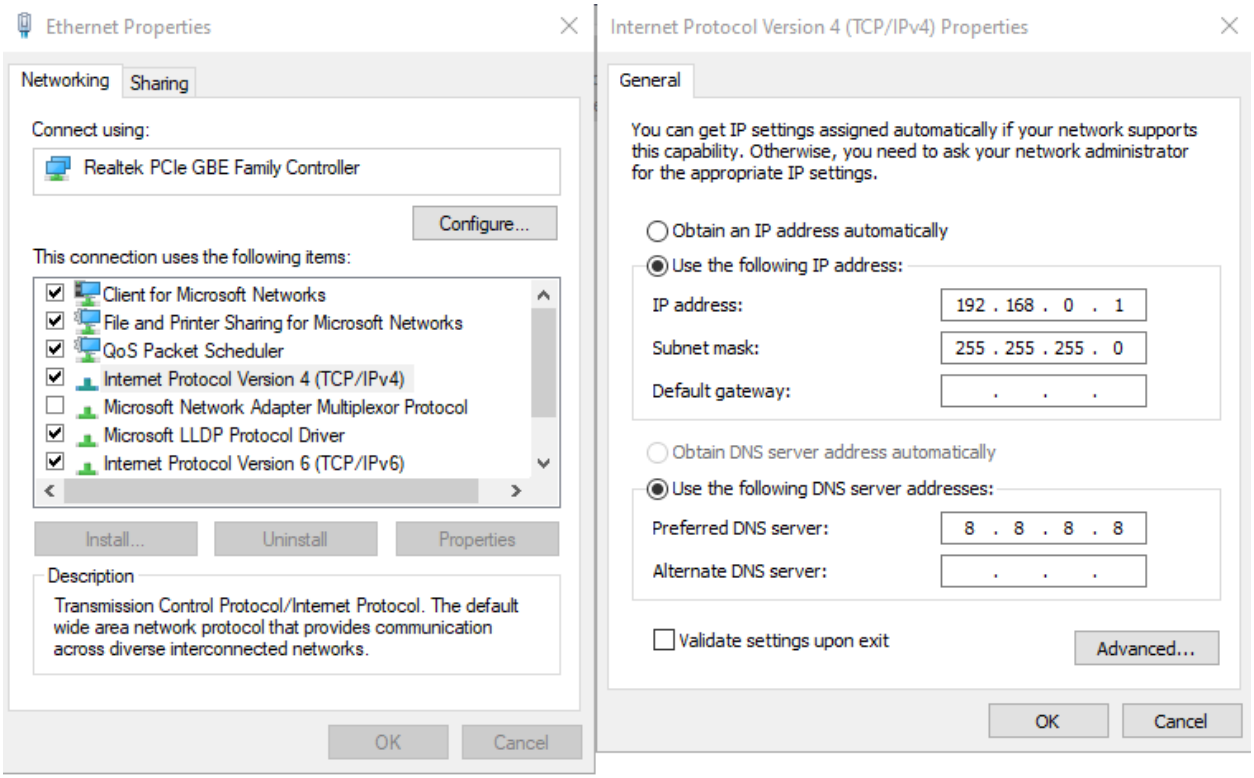
ET0104 Embedded Computer Systems Laboratory

Setting up the desktop for Ethernet connection :

Go to Wi-Fi Settings >> Ethernet >> Change adapter options

Right click on the Ethernet Computer and Go to Properties, click on Internet Protocol Version 4 (TCP/ IPv4) and change the IP address, subnet mask and DNS server according to the image below.





Setting up the Raspberry Pi with Ethernet Adapter :

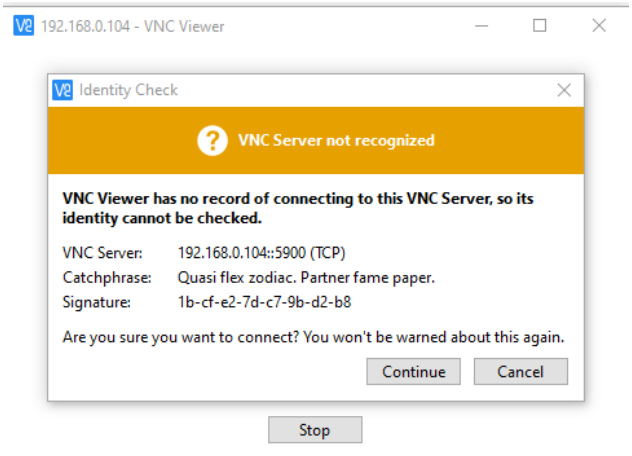
First, turn on the Raspberry Pi CM3 board.

Download VNC viewer:
https://www.realvnc.com/en/connect/download/viewer/

After installing VNC viewer, Run VNC viewer.

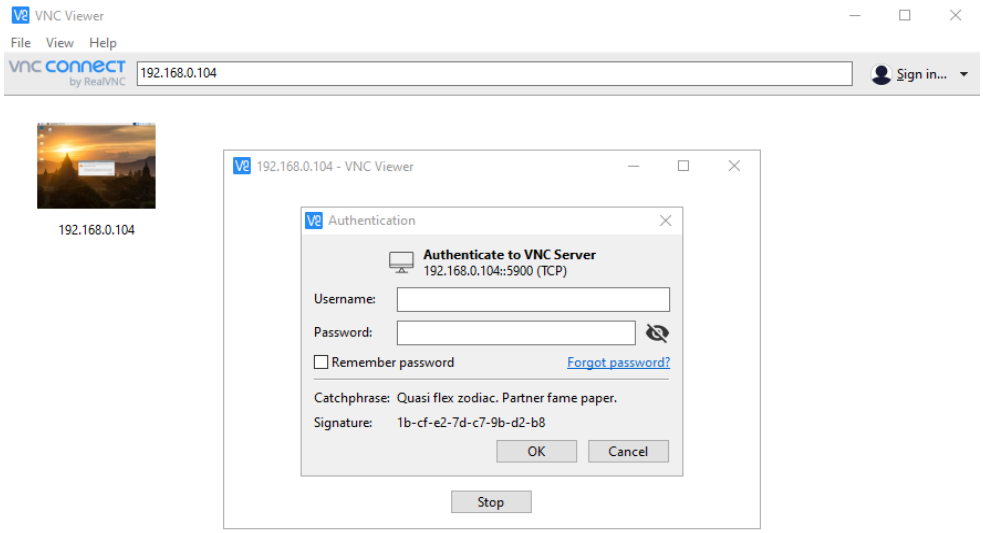
Type the VNC Server address “192.168.0.104” in the search box and press “Enter”.

This window will pop out, click on “continue”.



At this window, Fill in the User and Password.

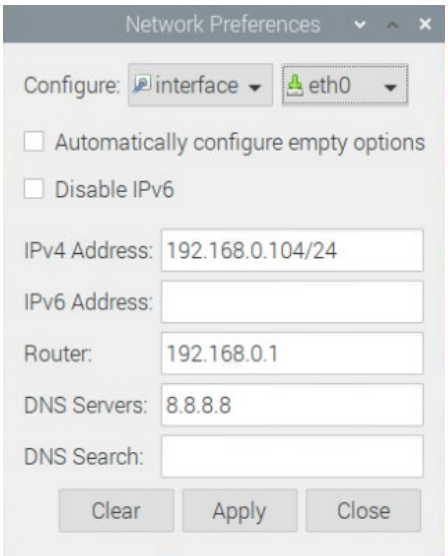
User : pi
Password : raspberry



Right Click on this icon  and click on Wireless and Wired Network Settings.

Click on the second box of Configure and select “eth0”.

Ensure that the IPv4 address, router address and DNS Servers are the same as the image below. Click on “Apply” and then click on “Close.”



Install gdbserver by opening terminal and command as follows:

sudo apt-get install gdbserver

To setup the Host Environment in Windows for the labs, the following SOFTWARE NEEDED:

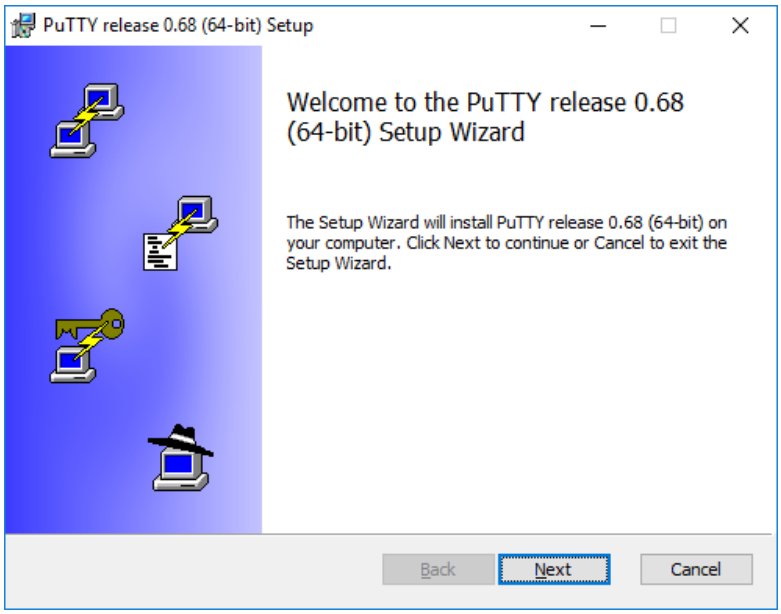
C:\PuTTY

PuTTY is a free implementation of SSH and Telnet for Windows and Unix platforms, along with an xterm terminal emulator.

Download PuttY – With plink:

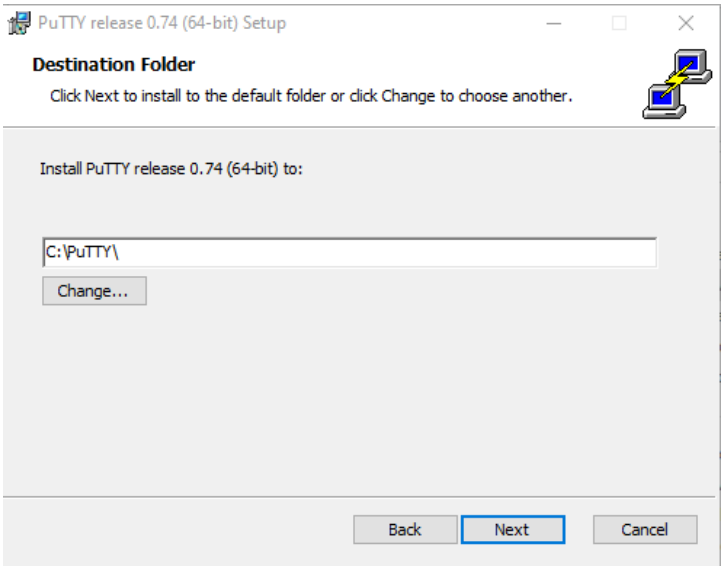
https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

Instructions for installing Putty:
When the installer starts, it shows the welcome screen. Just click Next

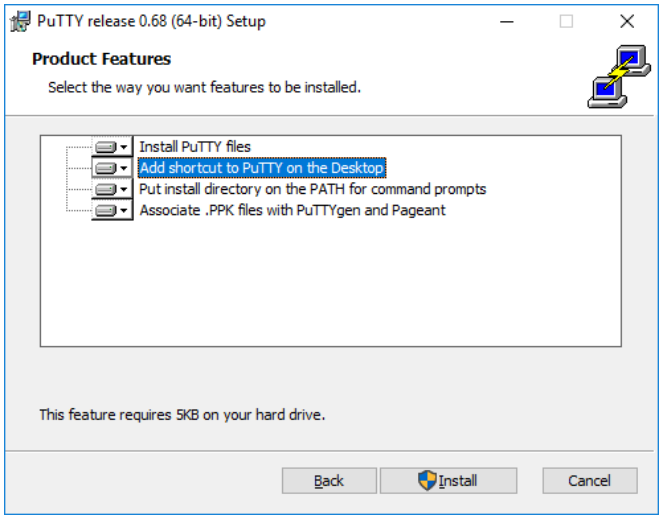


Then, the installer asks to select product features to install. You probably want to add a shortcut on the desktop if you expect to use the software frequently. All the other options generally should be enabled. When ready, click Install.

Install putty in C directory as C:\PuTTY\

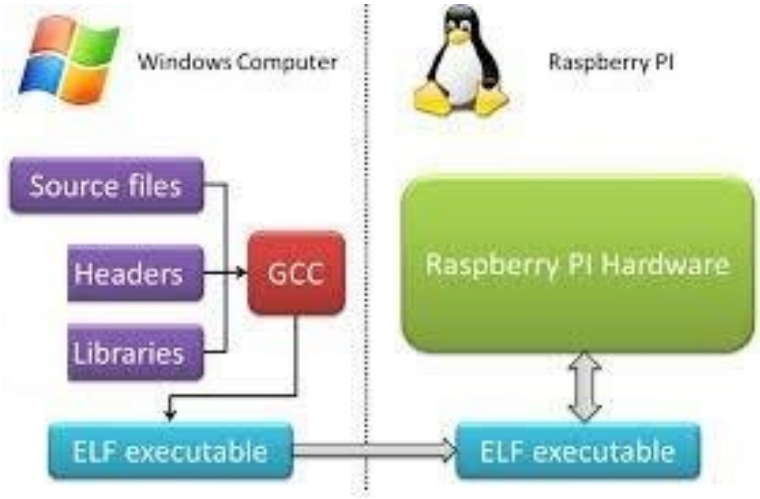


Then, the installer asks to select product features to install. You probably want to add a shortcut on the desktop if you expect to use the software frequently. All the other options generally should be enabled. When ready, click Install.



When the installation has completed successfully, it should show a "Completed" screen. Click Finish to exit the installer.

Cross Compiler Tool Chain for RPi on Windows



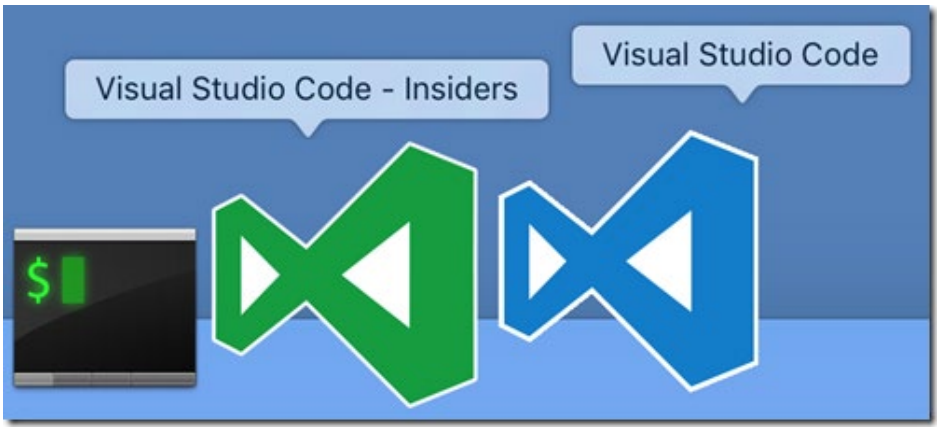
For embedded board running Debian-based GNU/Linux. This link provided complete toolchain for building and debugging Raspberry PI applications.

Each toolchain build includes the following components:
The GCC compiler for C and C++ languages
The GDB debugger
Include files and libraries from the compatible SD card image

Download : https://gnutoolchains.com/raspberry/download raspberry-gcc8.3.0-r2.exe

Instructions for installing cross complier:
* install cross complier toolchain for rpi in c directory
* check the installation is path as C:\SysGCC\raspberry

VS Code



Visual Studio Code is a free source-code editor made by Microsoft for Windows, Linux and macOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git.

Download VS Code :
https://code.visualstudio.com/insiders/

* In host open vscode workspace with files as

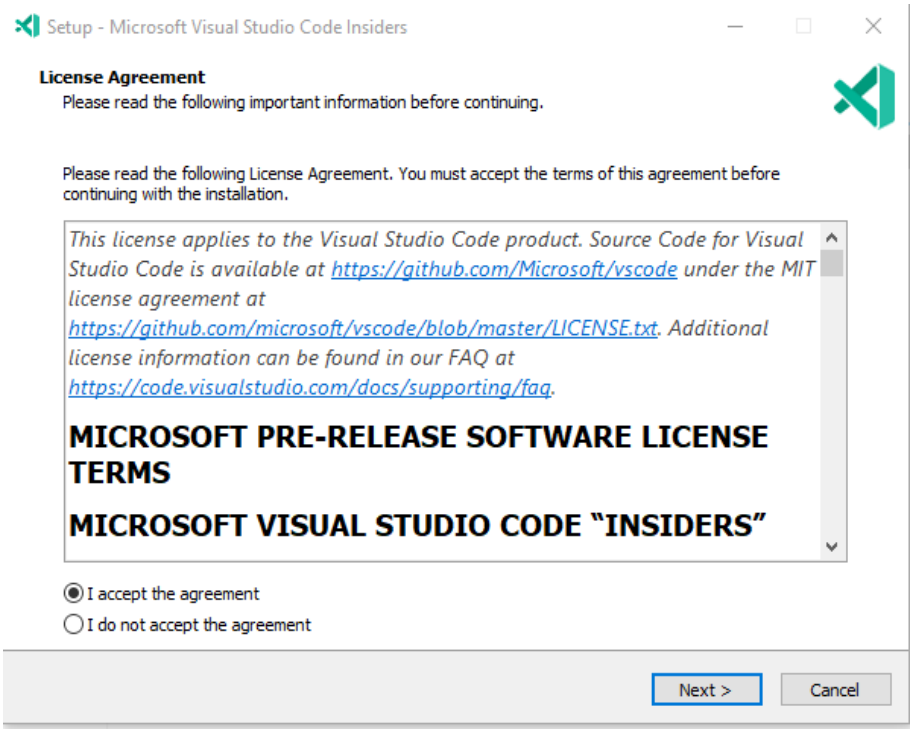
* lab1.c is a sample program showing how to control a 7 SEGMENT LED,

* library.c contains library functions for all source code,

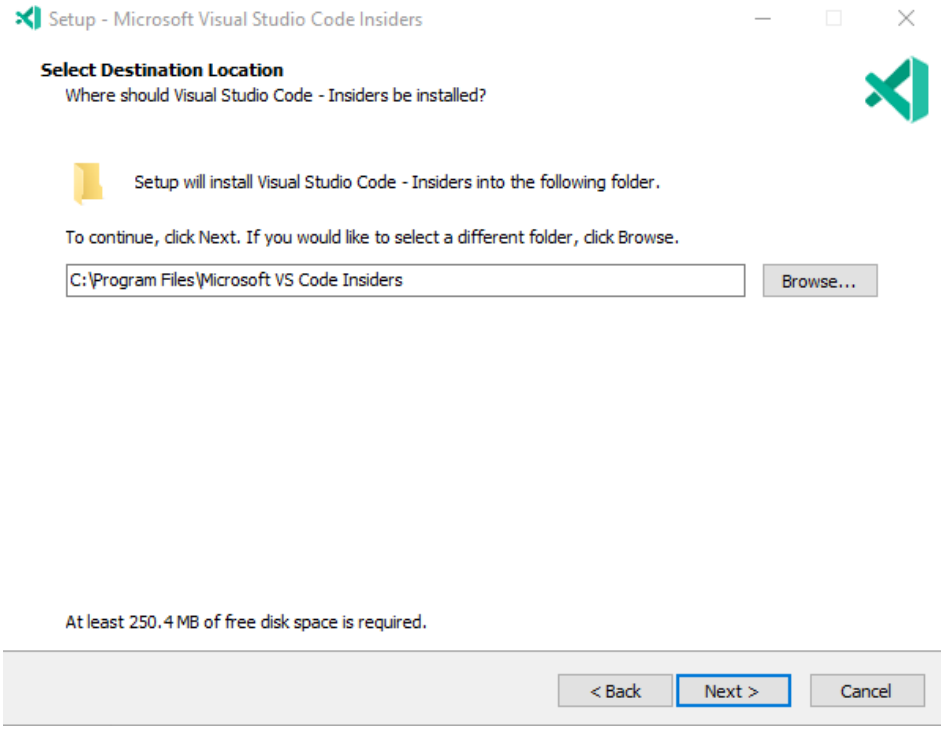
* library.h file contains function declarations and macro definitions to be shared between several source files,

* In VS Code press ctl+shift+p for opening the command palette,and type C/C++: Edit configurations (UI)

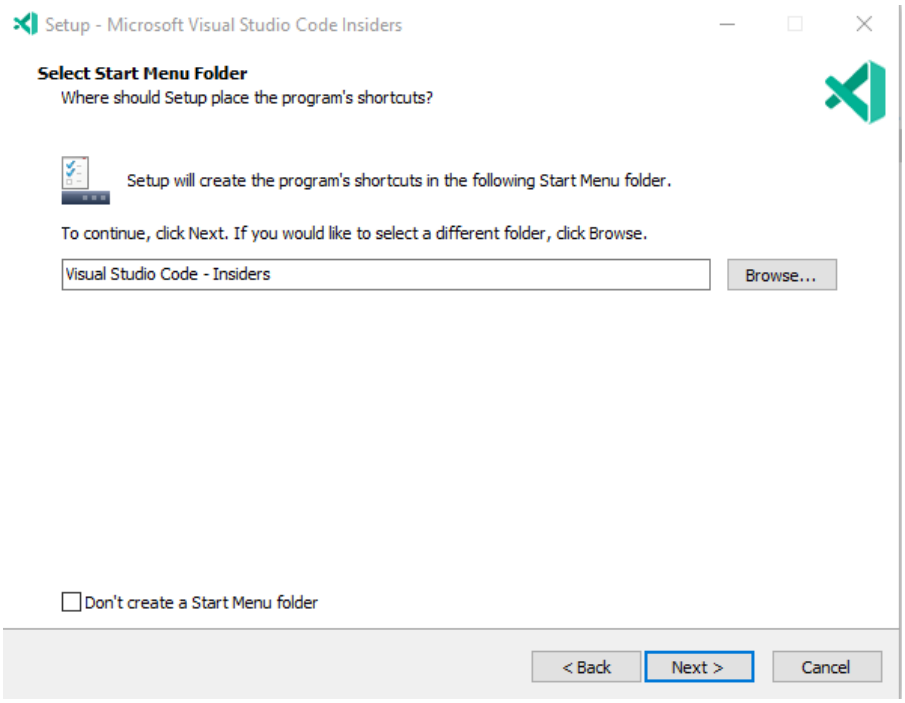
Click on “ I accept the agreement” and click on “Next” to continue.



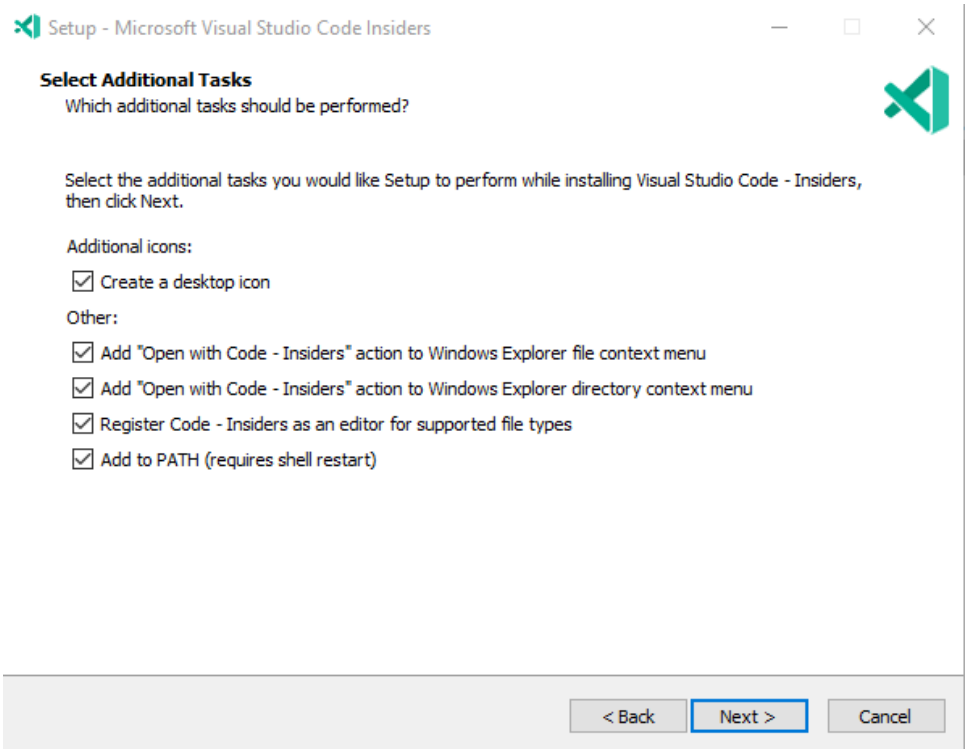
At this window, click on “Next” to continue.



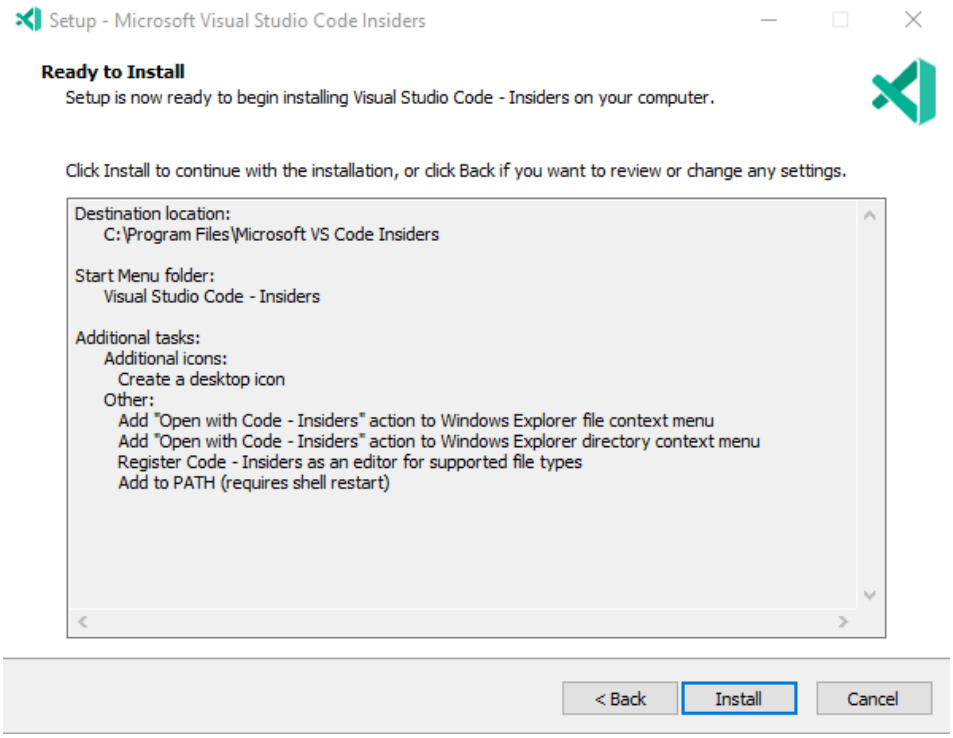
Click on “Next” to continue.



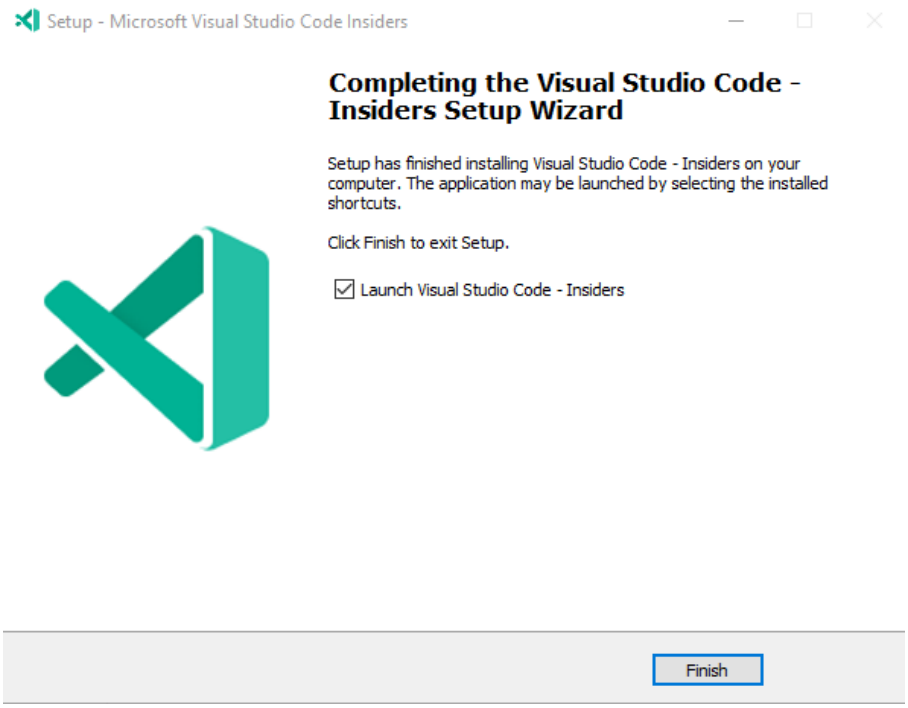
Ensure that the following boxes are ticked and click on “Next” to continue.



Click on “Install” to start the installation of Visual Studio Code.



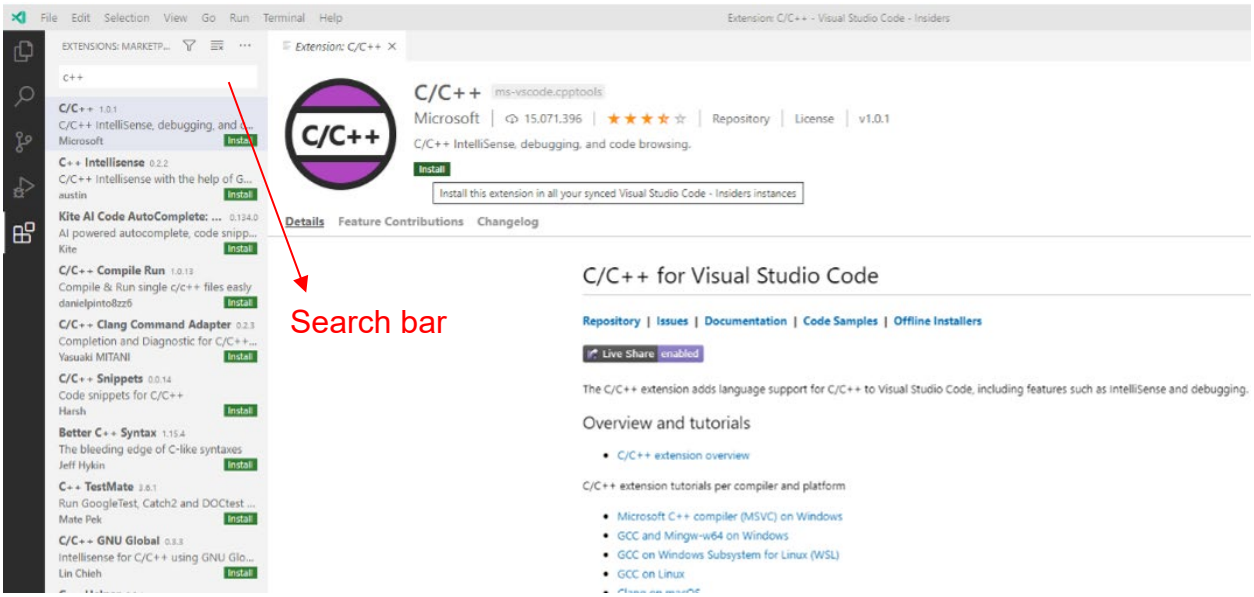
To launch Visual Studio Code, Click on “Finish”.



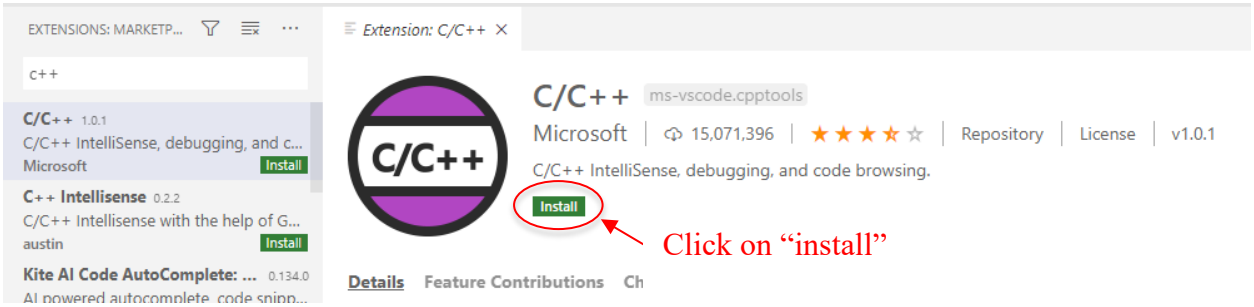
The extensions are to be installed :

- C/C++ 1.0.1
- ftp-sync 0.3.9

To install the C/C++ 1.0.1 extension, Click on the extension panel  and type in “C++” in the search bar.

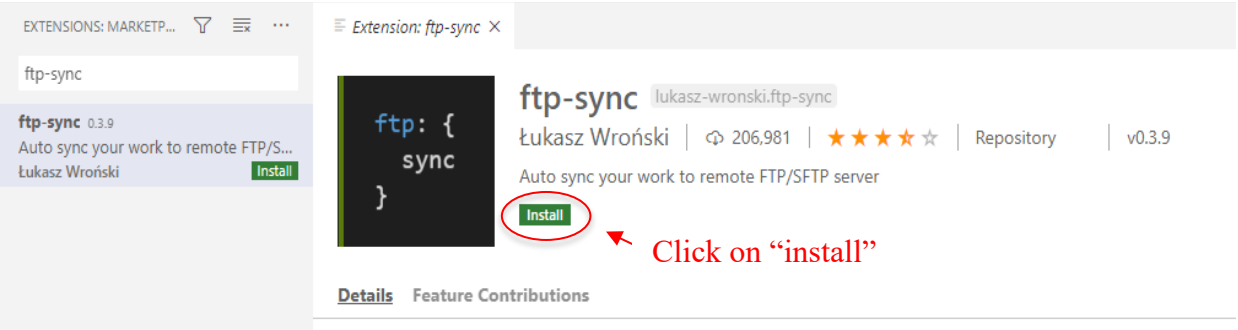


Click on the C/C++ 1.0.1 extension, click on the install button to install the extension.



To install the ftp-sync extension, Click on the extension panel  and type in “ftp-sync” in the search bar.

Click on the C/C++ 1.0.1 extension, click on the install button to install the extension.



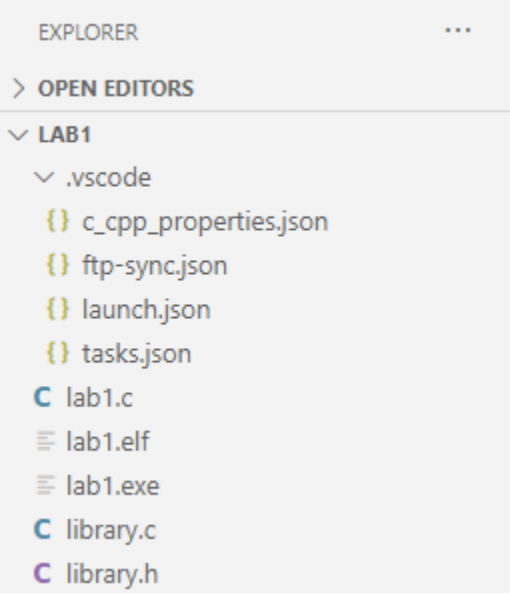
In the Visual Studio Code Insiders, click on “File” , then click on “Open Folder”.

Click on “This PC”, then click on “Win 10 (C:)”.

Click on the folder named “ECSLAB”.

Click on “lab1” and click on “Select Folder”.

The image is displayed below after doing the above instructions.

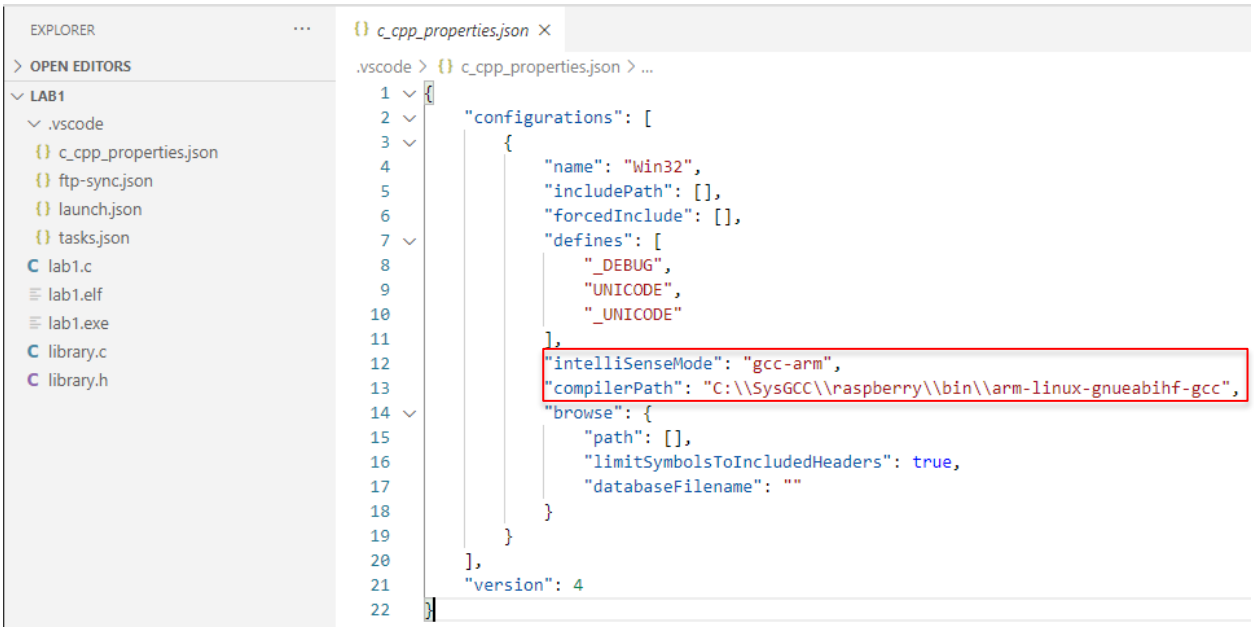


For c_cpp_properties.json file,

Click on the c_cpp_properties.json file and change the following command:

“intelliSenseMode”: “gcc-arm”,

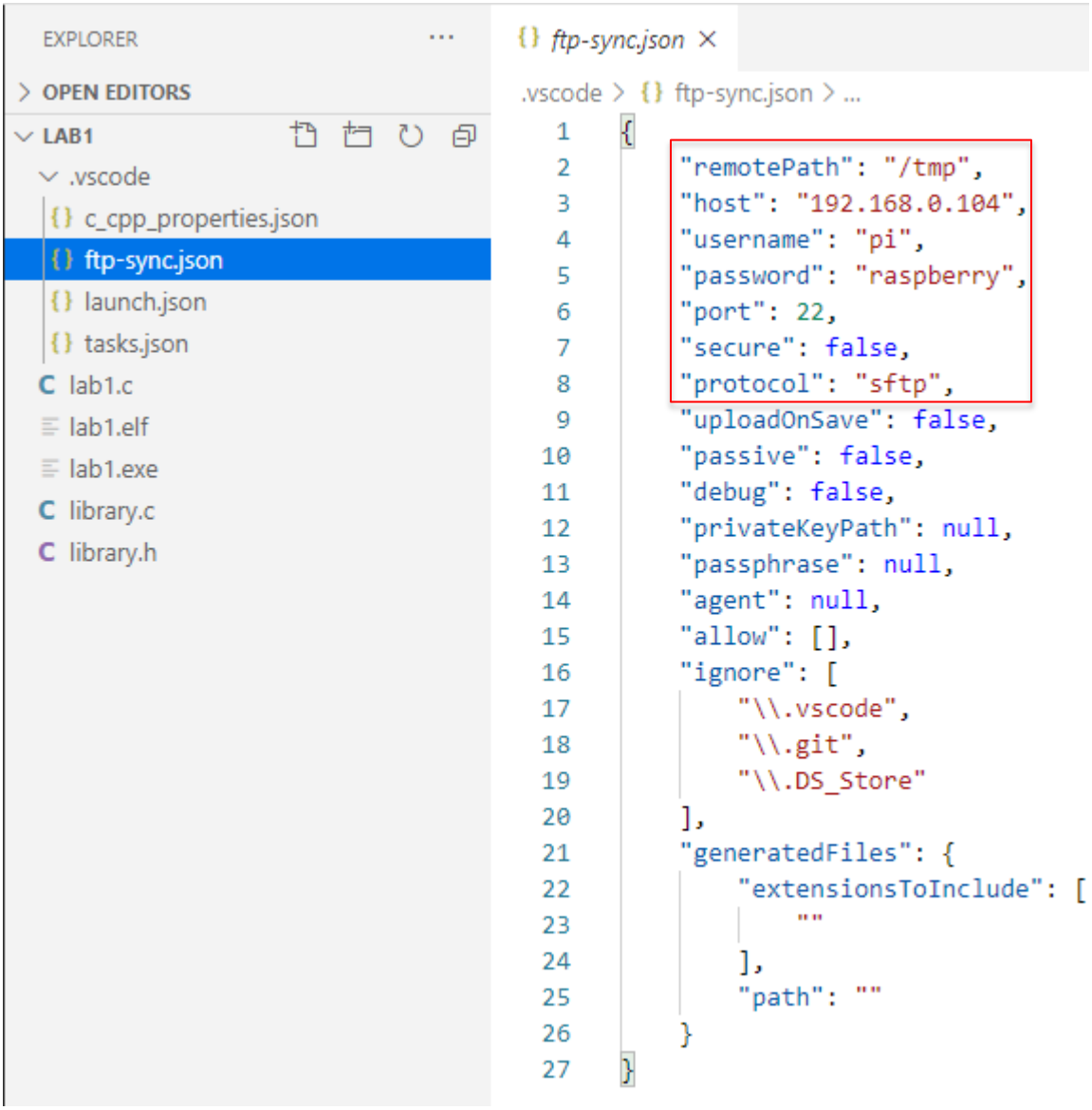
C:\SysGCC\raspberry\bin\arm-linux-gnueabihf-gcc



For ftp-sync.json file,

Click on the ftp-sync.json file and change the following command:

```
"remotePath": "/tmp",
"host": "192.168.0.104",
"username": "pi",
"password": "raspberry",
"port": 22,
"protocol": "sftp",
```



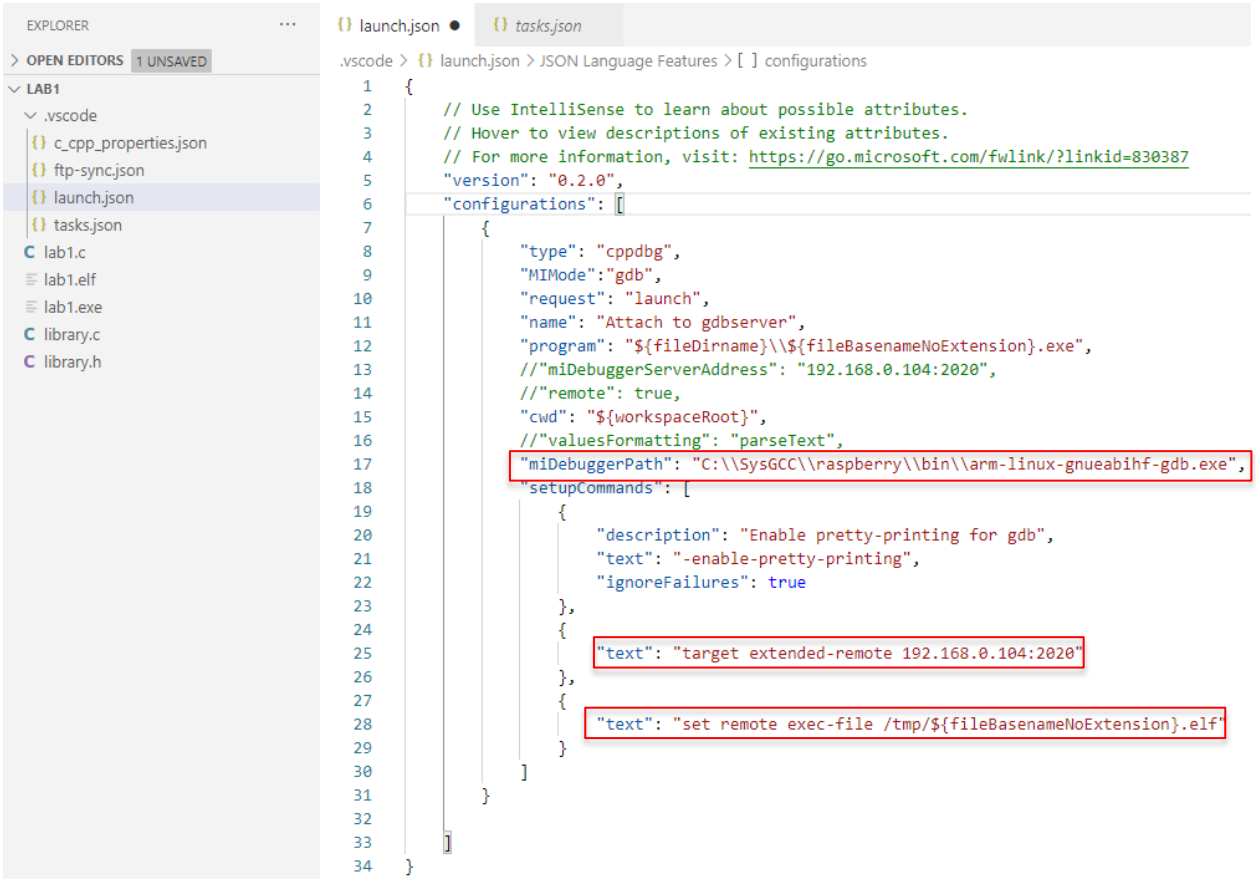
For the launch.json file,

Click on the launch.json file and change the following command:

"miDebuggerPath": "C:\\SysGCC\\raspberry\\bin\\arm-linux-gnueabihf-gdb.exe”,

"text": "target extended-remote 192.168.0.104:2020"

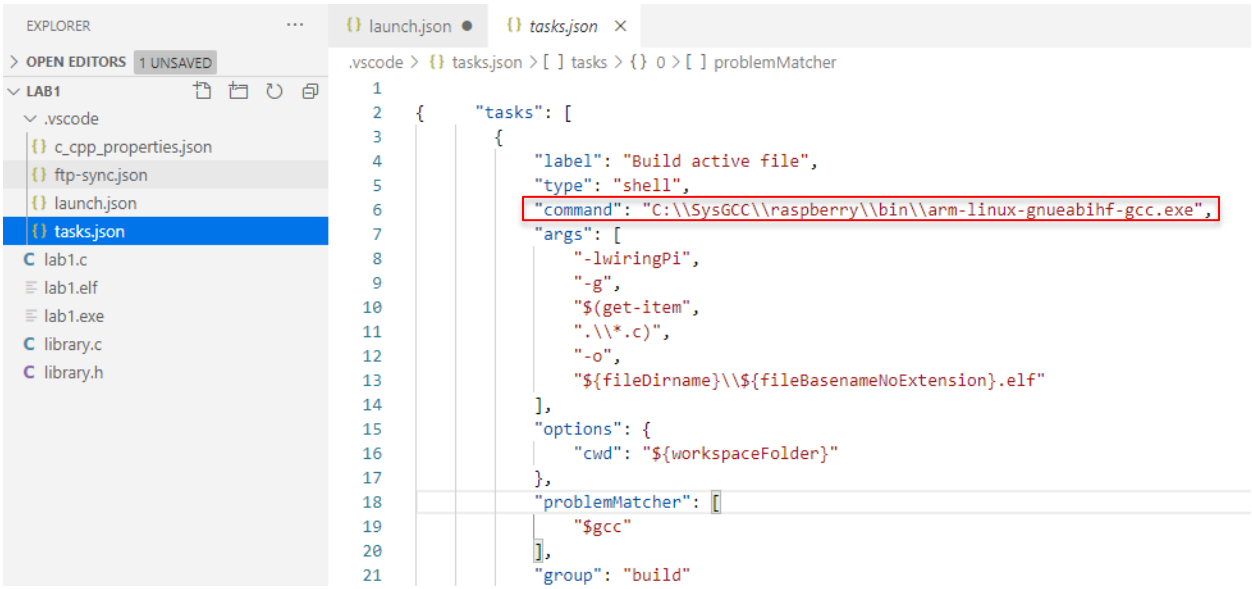
"text": "set remote exec-file/tmp/${fileBasenameNoExtension}.elf"



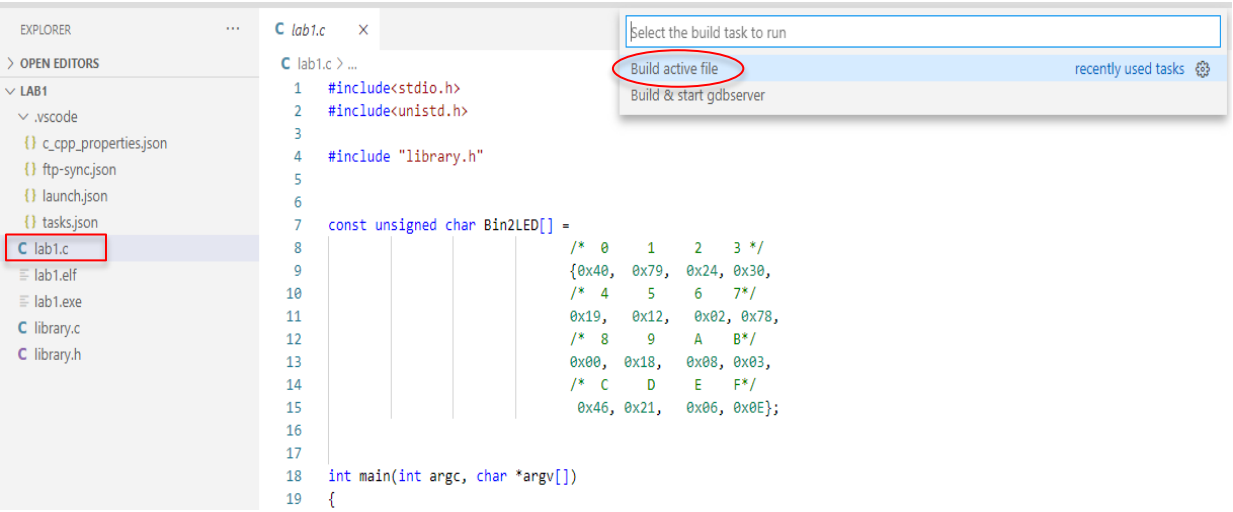
For the task.json file,

Click on the task.json file and change the following command :

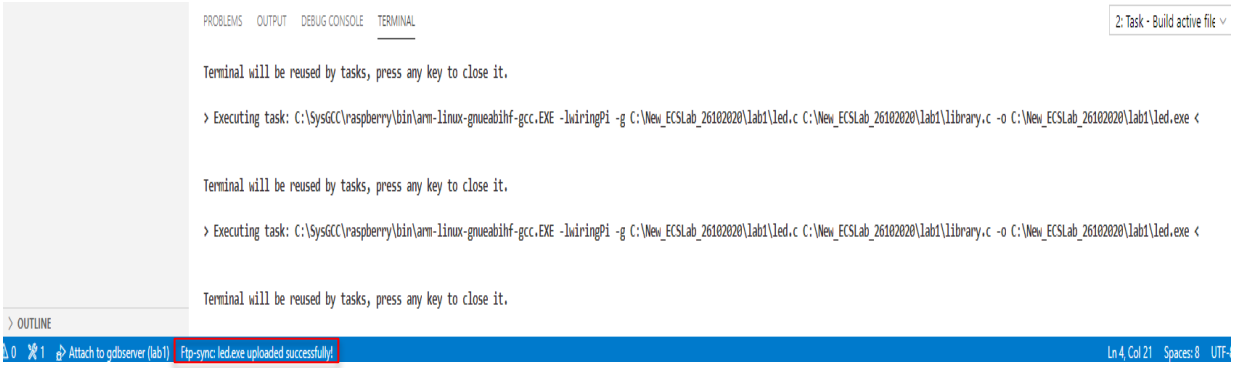
"command": "C:\\SysGCC\\raspberry\\bin\\arm-linux-gnueabihf-gcc.EXE",



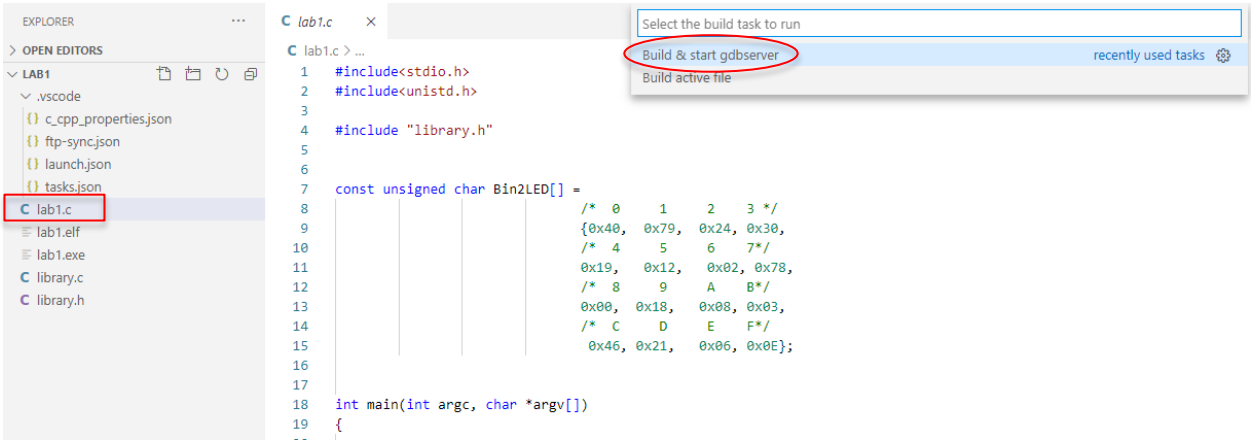
Click on lab1.c file and Ctrl-Shift-B and click on “Build active file”,



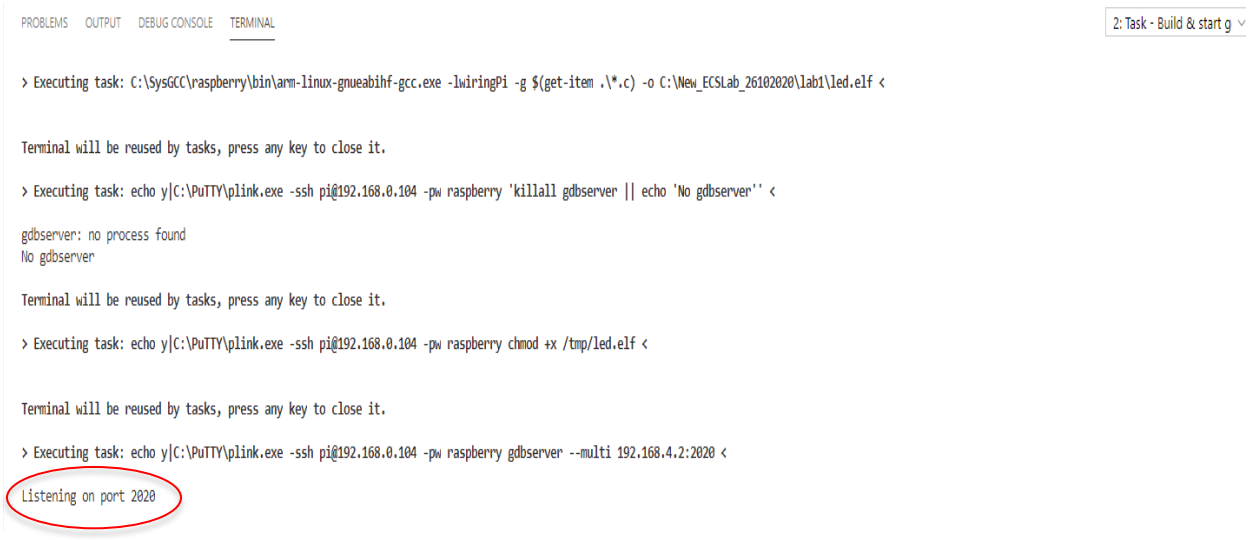
This will compile the program. In the terminal, the ftp-sync:lab1.exe will be uploaded successfully.



After "build" is finished, click on lab1.c file and Ctrl-Shift-B and click on "Build and start gdbserver" to run the gdbserver.



At the end of Terminal, there should be a "Listening on Port 2020" text.



To debug the program, Press the F5 key.

Now the LED will be displaying ‘0’ to ‘F’.

Troubleshooting

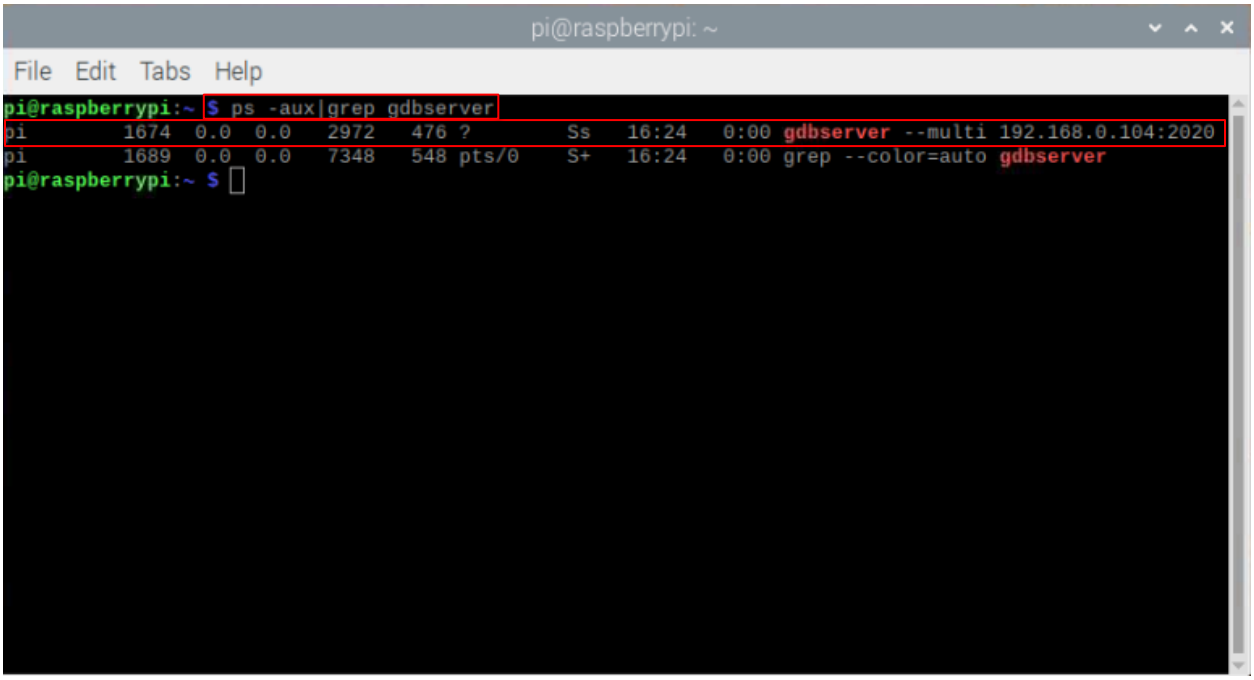
Click on the Raspberry Pi Terminal 

For checking if the gdbserver is running,

Enter the command:

pi@raspberry: ~$ ps -aux | grep gdbserver

The terminal shows that the gdbserver is running.

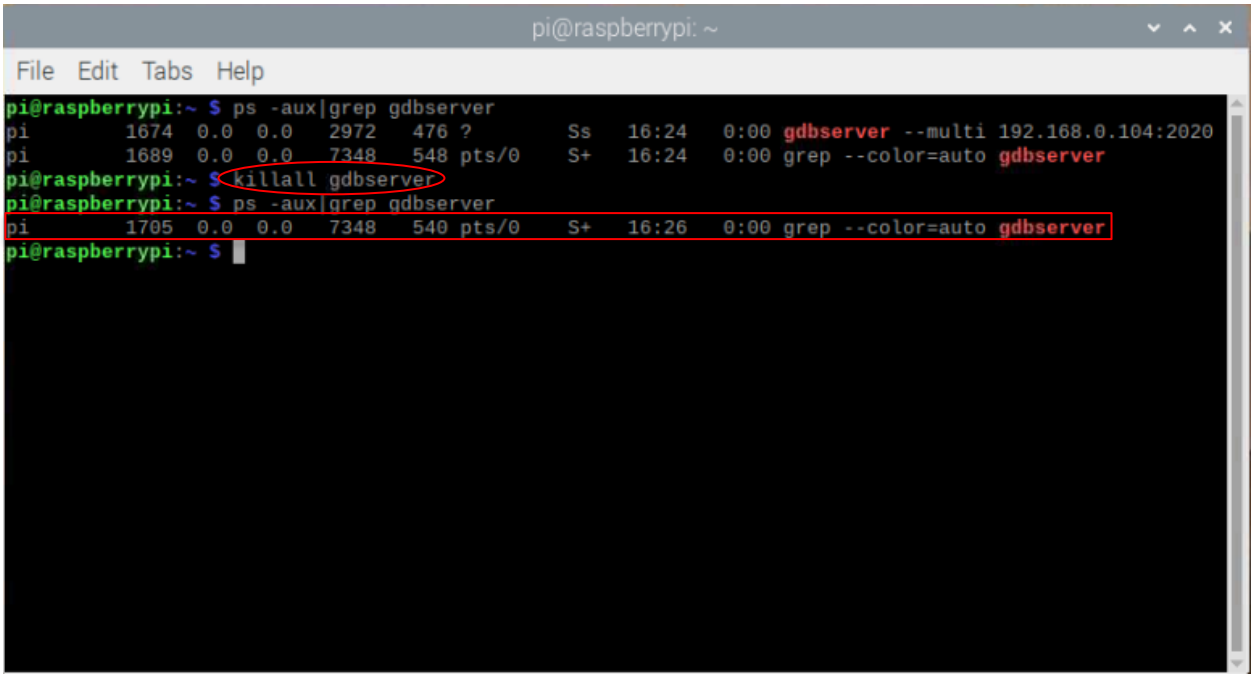


Note: Only use killall gdbserver when the address is being used or when the program needs to be stopped.

Enter the command :

pi@raspberry: ~$ killall gdbserver

After entering the command, there should be no gdbserver running.



SINGAPORE POLYTECHNIC
SCHOOL OF ELECTRICAL AND ELECTRONICS ENGINEERING

ET0104 Embedded Computer Systems Laboratory

Laboratory 1 Introduction to the Laboratory set up

1. Objectives

- To learn how to work with a single board computer (SBC) based on the Raspberry Pi and a personal computer (PC)
- To gain familiarity with an attached I/O board by writing a timer program.
- Learn how to create a project in Visual Studio Code and download it to SBC.

2. Introduction

The laboratory set up comprises of:

i) A PC running Visual Studio Code (VS Code)

ii) A Compute Module 3 (CM3) which is a single board computer (SBC) based on the Raspberry Pi. The SBC has all the facilities of a desktop computer. A larger list of its capabilities can be found in the appendix.

This computer runs on a version of the Linux operating system.

iii) A custom designed Input Output (I/O) board. This interfaces to the SBC through the bus interface to several pieces of hardware.

iv) Various I/O devices, like keypads, stepper motors, loudspeakers may be attached through connectors to this I/O board.

We will describe the board in some detail in the next section.

This board has four main areas of interest.

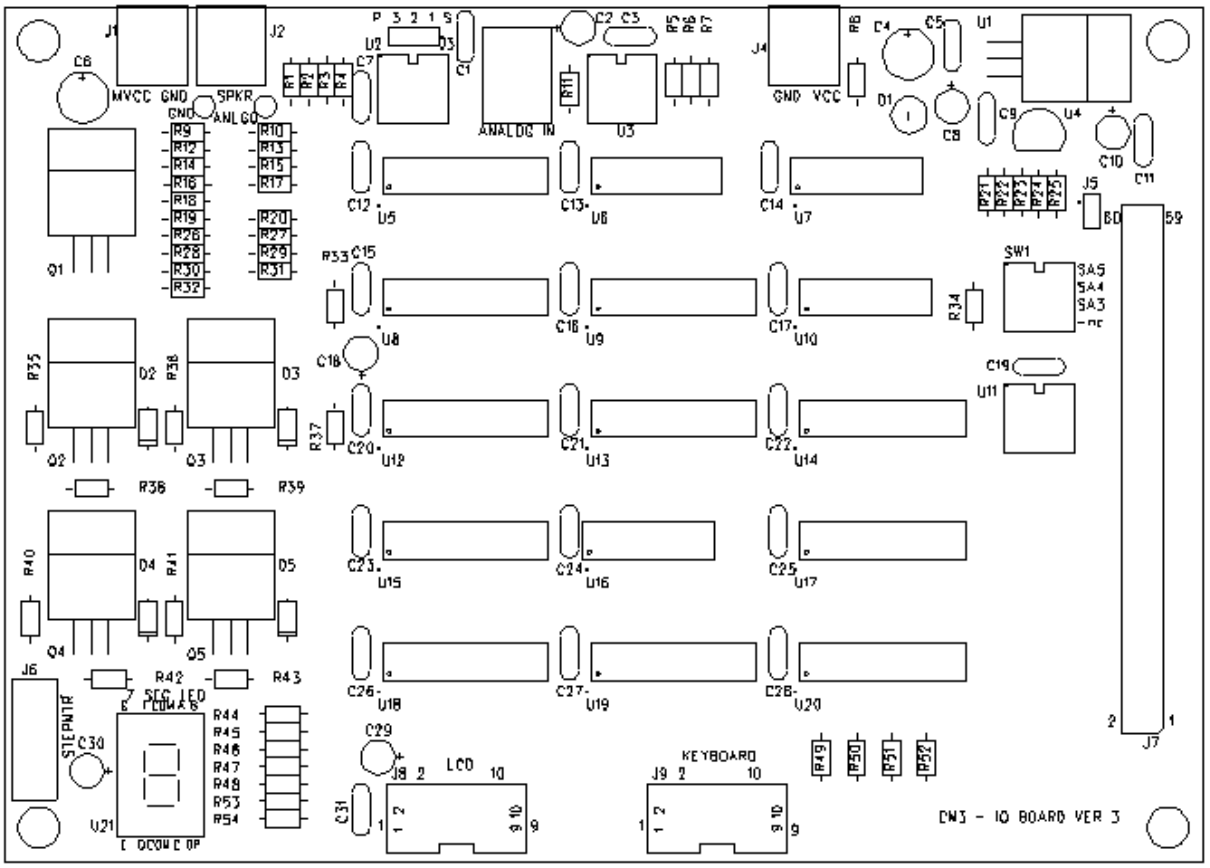


Figure 1 Layout of I/O board

First is the I/O connector, designated J7, located on the right edge of the I/O board. This 60 pin I/O adapter cable connects all the address and data busses and most of the control signals available from the Secondary Memory Interface (SMI) bus.

Second is the decoding switches, designated as SW1. These are located to the right of the board as well. They allow the top 3 address lines to be connected to a 74138 decoder on board. This in turn, will allow the user to access the 8 devices on board. We will consider them in greater detail later.

Third are the various buffers and latches. The board does have a single 7 segment LED for users to easily check out their programs initially. Fourth are the I/O connectors at the bottom edge of the board. These will be used to connect to various hardware devices. For example they are marked as“STEPMTR”, “KEYBOARD” and “LCD”.

Last of all are the analogue circuitry located at the top left of the board. This consists of a Digital to Analog Converter (DAC) made up of an R-2R resistor bank, fed from a 74LS244

buffer chip. This signal is buffered through an op-amp and is available as an Analog output marked “DA_OUT”. This may be connected to a speaker, which is buffered by a transistor. There is another DAC which is used for high speed data conversion connected via SPI (serial peripheral interface). The output can be selected by placing the jumper to P (parallel) or S (serial). Another SPI device, the Analog to Digital Converter (ADC) is connected to a variable resistor RV1 which allows a user to manually input a signal for testing.

Hardware block diagram

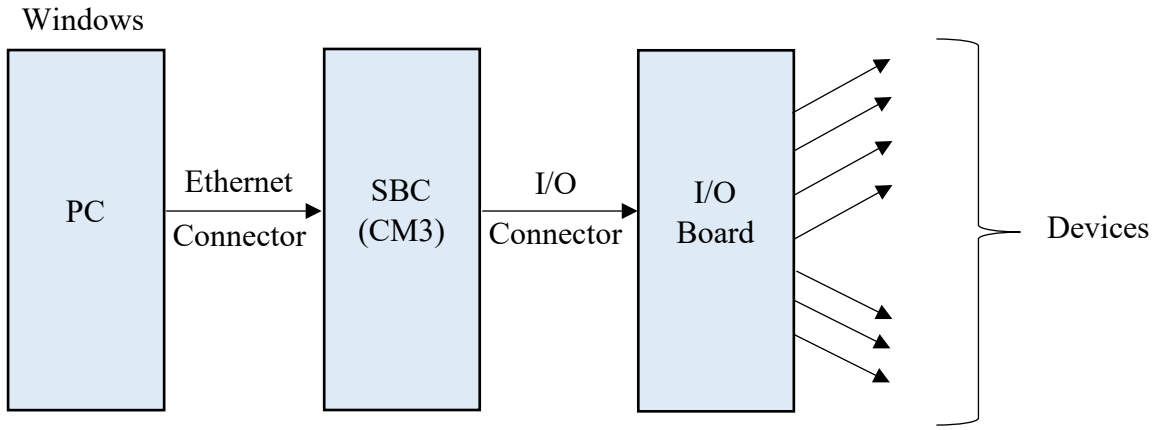


Figure 2: Overall block diagram

On the left we have a PC which in many cases runs on an Intel processor and Windows. This is connected to the SBC which is the CM3, an industrial version of the popular Raspberry Pi development system. This version has more I/O pins available for use and uses less peripherals so as to allow the user greater flexibility to configure their system.

The PC is connected to the SBC via an Ethernet connector for data transfer and control during the development process. In turn, the SBC connects to the I/O board via a 60 pin connector.

Software set up

We are working in a cross development environment, where the main development work is done on a computer with adequate resources – speed, memory and online connections. The source program is compiled on the host system and the compiled program is then transferred to the target system which is most probably running on another operating system and with much less resources.

In our setup, we are developing on a Windows platform and an Intel processor and executing the program on an ARM based system running Linux.

We use Visual Studio Code (VS Code), a free program editor which is created by Microsoft. This software is popular because it is very flexible, due its use of `json` (JavaScript Object Notation) files which are used to configure the particular working environment. In the C/C++ environment, VS Code uses the `task.json` and `launch.json` files for configuration. In the lab, we have configured VS Code to develop programs written in C for

the ARM (Advanced RISC Machine) processor used in the CM3 running on the Linux operating system.

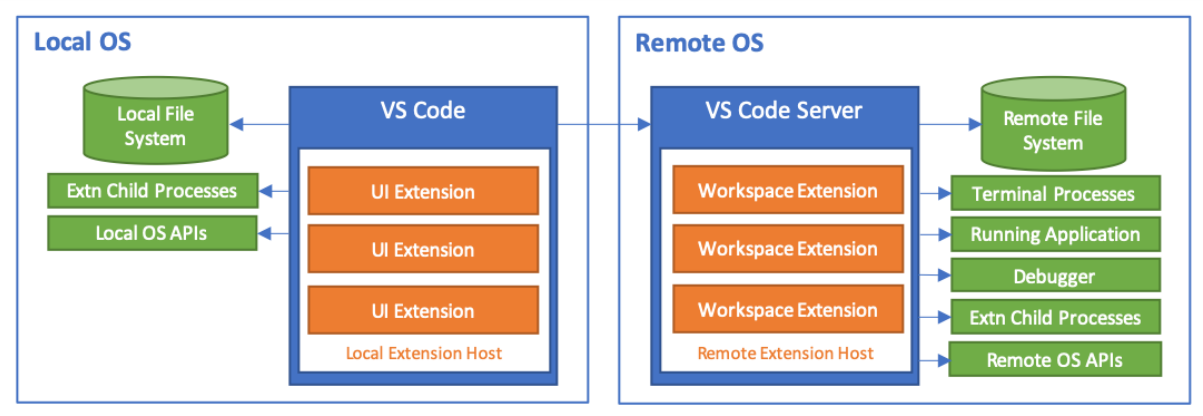


Figure 3 Block diagram of software environment

Host system - PC Intel running Windows – list of software installed and function	Target system – ARM / Linux – list of software installed
1. VS Code – develop program 2. GCC compiler for ARM and Linux 3. ftp-sync – transfer compiled file to target 4. gdbserver – execute / control program on target	1. Secure shell (SSH) enabled 2. gdbserver

3. Assignment

1) To gain some familiarity with the system, we will construct a timer which will update the time every 1 second. First let us review the operation of a seven segment LED. This is a device made up of seven light emitting diodes arranged so that when lighted appropriately, characters and numbers may be displayed. For example, the number "0" may be displayed by lighting up segments a to f. By showing the characters "A" to "F" as well, we can display one hexadecimal number. This is half a byte, or a nybble.

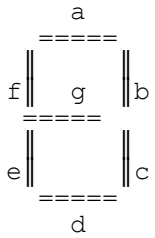


Figure 4: LED 7-segment display

2) It is a well-known fact that digital circuits in general are able to *sink* more current than they can *source*. An LED normally needs about 20 mA of current flowing through it before it lights up. Most TTL compatible circuits can sink this amount of current. Thus to turn *on* his LED, we output a *zero* level voltage. If connect each LED segment to a latch, we are able to display characters by outputting an appropriate byte to the latch. Let us utilise the following convention that latch pins 6,5,4,3,2,1,0 are mapped to segments g,f,e,d,c,b,a respectively. Complete the bit patterns for the following numbers. Remember, a "0" turns the segment ON.

   0: _____ 1: _____ 2: _____ 3: _____

   4: _____ 5: _____ 6: _____ 7: _____

   8: _____ 9: _____ A: _____ b: _____

   C: _____ d: _____ E: _____ F: _____

3) Since only the first four bits of a byte is used, we will need to "mask off" that is, set the high four bits to zero.

4) There are 8 hardware devices set up as follows. Most are latched output devices.
   Device 0 - Digital to Analog network
   Device 1 - Stepper motor
   Device 2 - LED
   Device 3 - Liquid Crystal Display (LCD)
   Device 4 - Lower 4 bits output to keypad / Higher 4 bits input from keypad
   Device 5 - Device not used
   Device 6 - Analog to Digital network
   Device 7 - Device not used
   Device 8 - Device not used

5) For this module, you will develop programs to control devices through this board, on a desktop PC. After that, you will download the program to the CM3, which will then interface to the I/O board.

4. Setting up the hardware

1) Make sure the power supply to the SBC is turned off.

2) We need to set up the I/O board so it can detect the address of the LED latch. Set the switches as follows. Setting a switch to “Off” places a logic “1" at the signal.

Note: Please use only your finger tips - do NOT use pens, or other sharp objects which may break off and leave debris on the I/O board.

Address	A5	A4	A3
DIP setting	Off	Off	On

Table 1 DIP settings for address

What this means is that for DIP switch block SW1, set the switch marked “A5" to the “Off” position. The same procedure applies for the other switches in SW1.

3) Now turn on the power supply to the CM3.

5. Working on files in a folder

1) As with other language development systems, a workspace refers to directories (or folders) that contains the source files. In VS Code, there is an additional `.vscode` directory which further specifies how the project files are to be manipulated through the use of `json` files.

   We will use a directory that has been created which makes up a project and modify the files within. We will use VS Code to access the lab1 file to create the necessary files for connect and transfer the necessary files to the CM3.

2) Click on the VS Code icon.

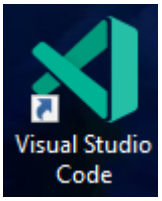


Figure 5: VS Code icon

6. Developing a program

1) In VS Code, select ***File*** > ***Open Folder*** > Win 10 (C:) > ***ECSLAB*** >
   Click once on ***lab1***.

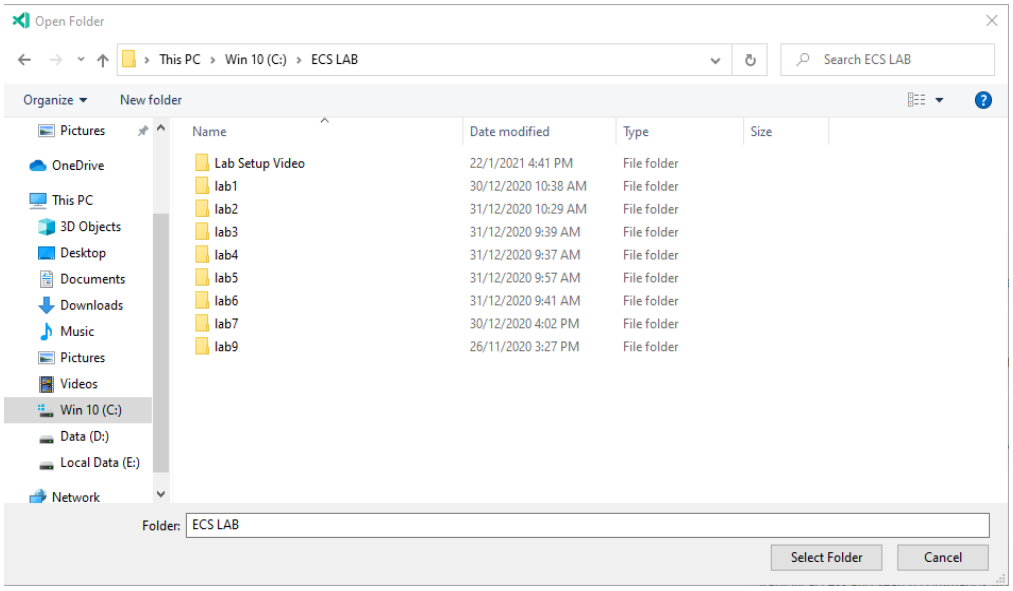


Figure 6: Opening a project folder

2) This will load the files for lab1 into VS Code. On the left side of the screen, the files in the folder will be displayed as shown in Figure 7.

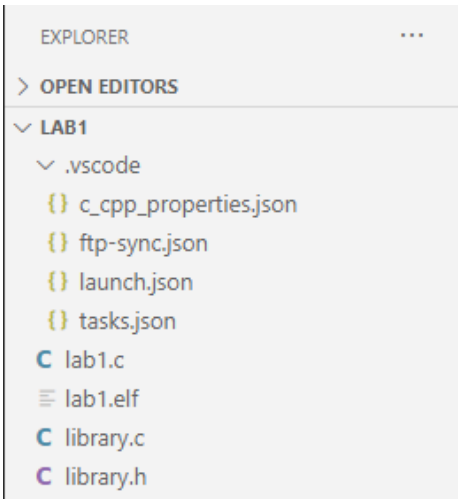


Figure 7: List of files in the lab 1 folder

7. Compiling and uploading the program

1) In this step, we are performing cross compiling a program meant for another processor on a PC. This executable file will have an elf (executable and linkable file) extension and will only run on *that* target platform.

2) Select **lab1.c** (Box 1), press Ctrl-Shift-B and click on ***Build active file*** (Circle 1) shown in Figure 8. This will compile the C program into an elf file. The system is configured

so that the ftp-sync function will automatically upload the `elf` to the target system. The status of this is shown at the bottom of the screen in Box 2 of Figure 9.

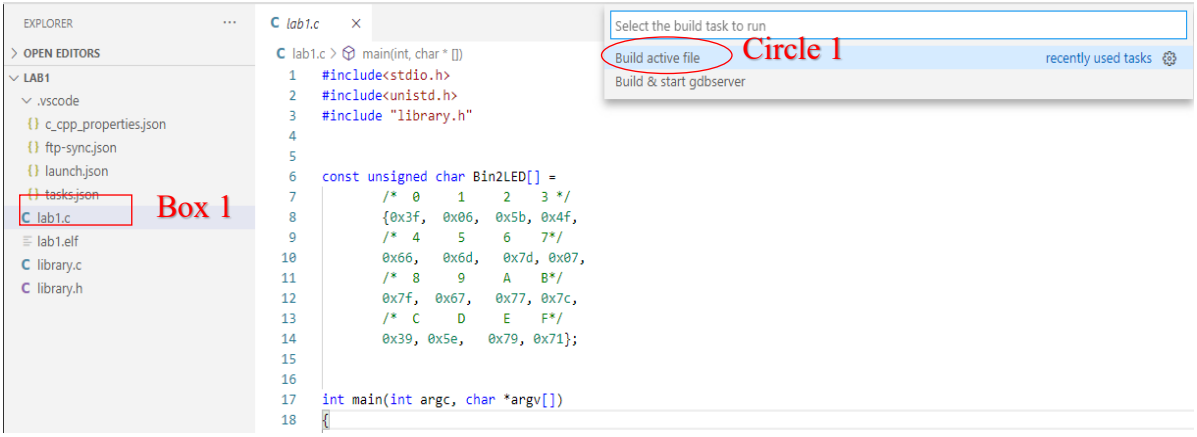


Figure 8: VS Code screen

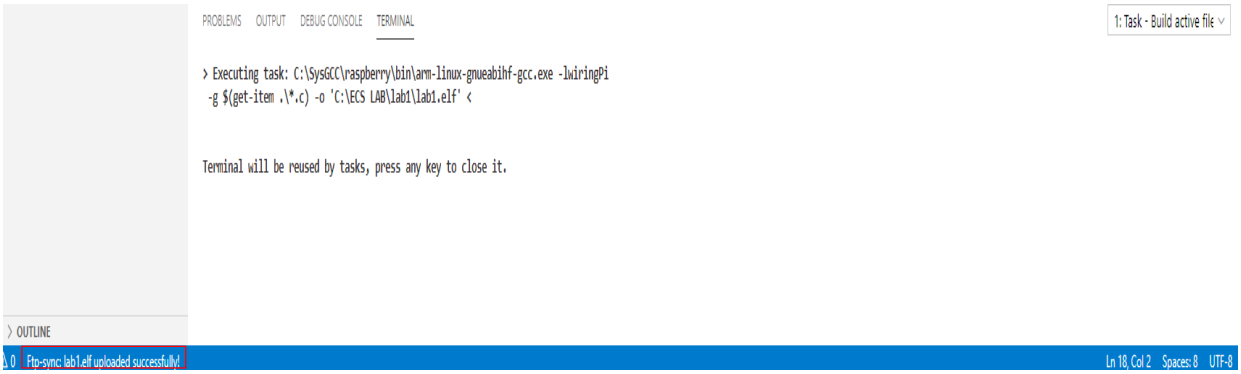


Figure 9: Bottom of VS Code screen

If there are any errors, correct them and build again.

3) After a successful build, click on **lab1.c file** (Box 3) then Ctl-Shift-B and click on ***Build and start gdbserver*** (Circle 2) to run `gdbserver`, which will execute the elf on the CM3 in Figure 10.

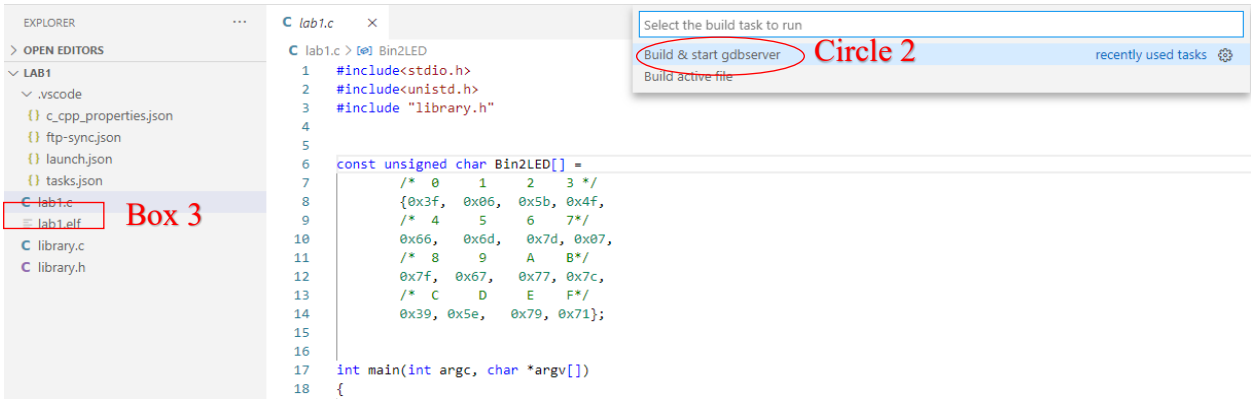


*Figure 10: Procedure to **Build and start gdbserver***

4) At the bottom of the screen, there should be a message `Listening on Port 2020` (Circle 3) which indicates that the program is ready to run.

Figure 11: The status of "Build & start gdbserver"

8. Executing the program and power down

1) To execute the program, Press the F5 key. At the bottom of the screen, the status of the program is shown in Figure 12. This indicates that the program executes successfully. You can view the status of the program in the Terminal and Debug Console tabs.

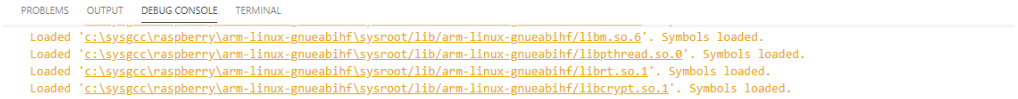


Figure 12: The status of the program after executing it

2) The LED will display '0' to 'F' at one second intervals.

In case of errors

If there are problems, you may modify your program or change the DIP switch settings.
If you need to change your DIP switch setting, it is highly recommended that you stop the CM3.

Power off

The CM3 just like the Raspberry Pi, uses a full operating system. It needs to be shut down properly. The lab is set up so that this can be done from VS Code.

From the VS Code menu, press Ctl-Shft-P (or View > Command Palette) and at in the edit box, type ***Tasks: Run Task*** and press Enter.

You will see a drop down list of commands. When you are ready, click on Power Down

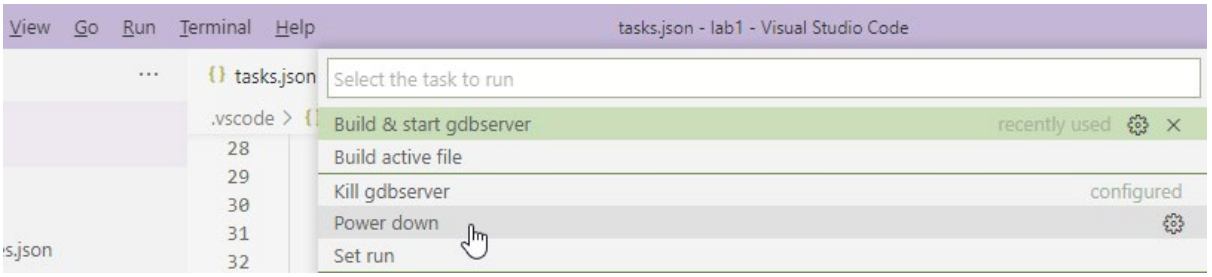


Figure 13 Power off CM3

Wait till the green LED stops flickering on the CM3 and then power can be removed.

Appendix 1

Program template for lab 1:

```
/***********************/
/*       LED lab       */
/***********************/
#include<stdio.h>
#include<unistd.h>
#include "library.h"
#define LEDPort 0x32

/* store this table in code space */
const unsigned char Bin2LED[] =

    /*  0     1     2    3 */
     {0x__, 0x__, 0x__,0x__,
    /*  4     5     6    7*/
      0x__, 0x__, 0x__,0x__,
    /*  8     9     A    B*/
      0x__, 0x__, 0x__,0x__,
    /*  C     D     E    F*/
      0x__, 0x__, 0x__,0x__
    };

/*********************************************************************/
/**********************      MAIN PROGRAM      **********************/
/*********************************************************************/
     int main(int argc, char *argv[])
{
     int LEDval;
     CM3_DeviceInit();

    for(int i=0;i<16;i++)
    {
        LEDval = Bin2LED[i];
        CM3_outport(LEDPort,LEDval);
        printf(" %d \n",i);
        sleep(1);        // non blocking sleep
    }
   CM3DeviceDeInit();
}
```

Appendix 2

Some considerations for developing programs for Linux on a Windows platform

Visual Studio Code Remote Development allows you to use a container, remote machine, or the Windows Subsystem for Linux (WSL) as a full-featured development environment.

The Windows Subsystem for Linux lets developers run a GNU/Linux environment - including most command-line tools, utilities, and applications -- directly on Windows, unmodified, without the overhead of a traditional virtual machine or dual-boot setup.

You can:

- Develop on the same operating system you deploy to or use larger or more specialized hardware.
- "Sandbox" your development environment to avoid affecting your local machine configuration.
- Make it easy for new team members to get started and maintain a consistent environment.
- Use tools or software not available on your local OS or manage multiple versions of them.
- Develop your Linux-deployed applications using the Windows Subsystem for Linux.
- Access an existing development environment from multiple machines or locations.
- Debug an application running somewhere else such as a customer site or in the cloud.

Some specifications of the CM3 as a Single Board Computer:

1. A credit-card sized computer that plugs into a computer monitor or TV.
2. The main CPU is the System on Module which is latched to the base board.
3. The baseboard is one which will connect to the outside world. It contains following:
    a. One USB Port
    b. One HDMI Port
    c. 120 GPIO pins.
4. 4GB eMMC Flash, 1 GB RAM
5. Runs Linux based Raspberry Pi OS (previously called Raspbian)

Laboratory 2 Further features of VS Code and C

1. Introduction

In this session, we will use some of the more powerful features of VS Code and its debugging environment. Specifically, we shall learn:

i) How to set breakpoints and examine variables.

ii) How to set watches.

2. Objectives

- To explore the debugging features of Visual Studio Code.
- To use the features of Visual Studio Code that encourage modular programming

3. Assignment

We will revisit the timer program used in the previous lab. Open the folder ***lab2*** in the subdirectory ***ECSLAB/lab2***. ***Click*** on the file ***lab2.c*** and fill in the blanks as before.

4. Frequently encountered problems in program development

1) We look at some frequently encountered problems:

i) Why did the program not execute at this part?

ii) Why did the value change/ not change?

2) The first question comes about after the program makes a decision, for example an *if*, *switch* or *while* statement. For this, we want to stop the program just after the decision. We also want to examine the test variable.
3) In the second case, we normally call a function to perform an operation - perhaps to output to a device. Again, we want to stop just before the function. Then we want to execute just one step after the procedure, or step *into* the procedure to see what went wrong by viewing the offending variable(s) as we execute the function one step at a time.

5. Breakpoints, stepping and watches

1) The point at which a program stops is called a "breakpoint". A breakpoint that can stop a program as it is executing at full speed without modifying the program itself is a "hardware breakpoint". As you may have guess, many debuggers insert a special instruction into a program to make it stop. This is a "software breakpoint".

2) When a program executes one line at a time, it is a "step". To observe the values of a variable as we step is to "watch" it. Looking at the lab2.c, let's say we want to see how the program converts the variable `i` to its 7 segment form. Also, we want to see the value being output to the port.

```
for(int i=0;i<16;i++)
{
      LEDval = Bin2LED[i];
      CM3_outport(LEDPort,LEDval);
      printf(" %d \n",i);   /* only for debugging */
      sleep(1);             /* non blocking sleep */
            }
```

We want to *break* at the statement: `LEDval = Bin2LED[i];`
After that, we want to step through and *watch* the values `i` and `LEDval`.

6. More detailed look at VS Code and debugging features

In this section, we look more into VS Code and explore some of its debugging features. The following diagram shows the main areas of the VS Code screen and we focus on those areas which will be used in this lab.

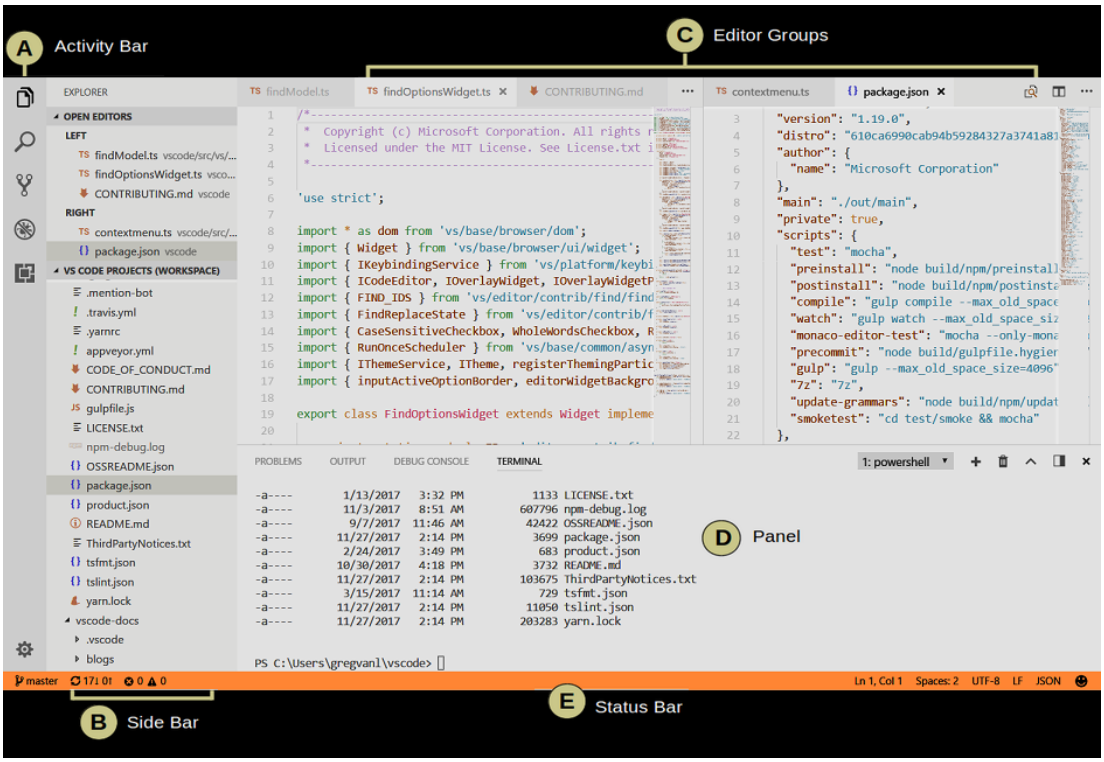


Figure 1 VS Code main user interfaces

1) Functions of the VS Code activity bar

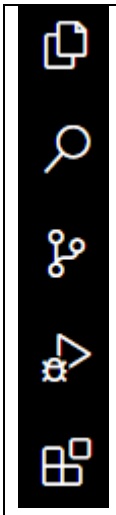
	Explorer - Used to browse, open, and manage all of the files and folders in your project.
	Search - Provides global search and replace across your open folder.
	Source Control - VS Code includes Git source control by default.
	Debug - VS Code's Debug View displays variables, call stacks, and breakpoints.
	Extensions - Install and manage your extensions within VS Code.

Figure 2: Description of VS Code activity bar

2) Previously we pressed the F5 key for the program to run immediately and terminate. Now we want to add a breakpoint to pause the execution at a certain point.

To do so, go to the *Activity Bar*, select the *Editor* icon. Select the file `lab2.c` in the file list in the *Side Bar* (or at the tab in the editor groups). Position the cursor at the statement:

```
LEDval =Bin2LED[i];
```

3) Press ***F9***, a red circle will appear next to the statement as shown in Figure 1.

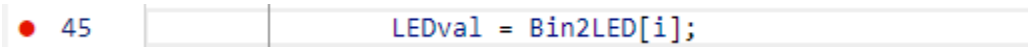


Figure 1: Breakpoint on the statement

4) To debug, we have to:

i) Compile the code which auto transfers the executable to the target system and then start the debugging system.
This is done by pressing Ctl-Shft-B. Click ***Build & start gdbserver*** – the Terminal window should show as confirmation:

```
Listening on port 2020.
```

ii) Press **F5** to start debugging. The debugging taskbar shown below will appear:

Figure 2: Debugging taskbar

5) Here is an explanation of the icons

Continue – equivalent to pressing F5.
Function: To continue running the program.

Step Over – equivalent to pressing F10.
Function: Run a function, execute it and continue at the next line.

Step Into – equivalent to pressing F11
Function: Command and advances the program execution one statement at a time.

Restart – equivalent to pressing Ctrl +Shift + F5
Function: Start the program at the main( ) function.

Stop – equivalent to pressing Shift + F5
Function: Stop the program from executing.

Step Out – equivalent to pressing Shift+F11

Function: Continues running code and suspends execution when the current function returns.

6) On clicking the Debug icon, the *Side Bar* will show the *panel* displaying the Variables (Box 1) will lists *all* the variables in the program - which could be very long. The Watch and Call stack panels are also displayed.

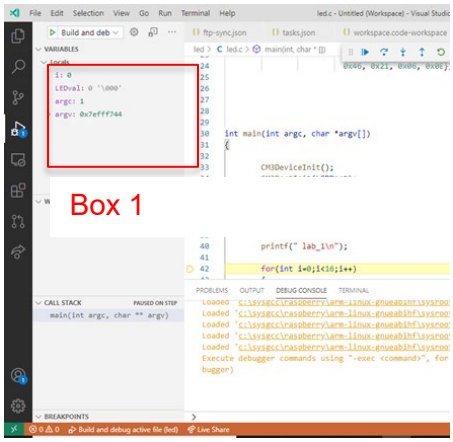


Figure 3 Local variables panel

7) To select only certain variables to *watch*, go to the Watch panel and position the mouse to the + icon. We want to watch `LEDval` and `i`. You need to type in these variable names - *and* you can set the number base of a watched variable.

For example to watch LEDVal in hexadecimal, type the expression `LEDVal,h` at the watch window.

Figure 4 Watch edit box

Another way of doing this is to right click the variable from the Local variables panel. Click on ***Add to Watch*** to bring the variable into the Watch panel. To display the value in hexadecimal, right click on the expression and ***Edit Expression*** to make the value display in hexadecimal as before.

i) Now, step through the program : ***Debug*** > ***Step Over (F10).*** Note the values of `LEDval` and `i` change as you step through the program.

ii) The basic breakpoint has now been set. However, the other breakpoint features can be very useful. We will now look at one of these features.

iii) You'll need to stop debugging: ***Debug*** > ***Stop Debugging (Shift-F5)***.

8) Editing the breakpoint - say we want to break only *after* a certain number of events happen. This is useful especially if something goes wrong only after several thousand runs! This is done by setting a conditional breakpoint.

i) ***Right-click*** on the breakpoint you set up – at the red dot. A drop down list will appear – select ***Edit breakpoint*** and enter the expression `i==9` so the program breaks only when i becomes 9. Press ***Enter*** to confirm.

ii) Start debugging again and note the value of `i` when it does break. You can also *restart* the debugging session.

9) You may wish to try out other ways of enabling the conditional break, or the other break options.

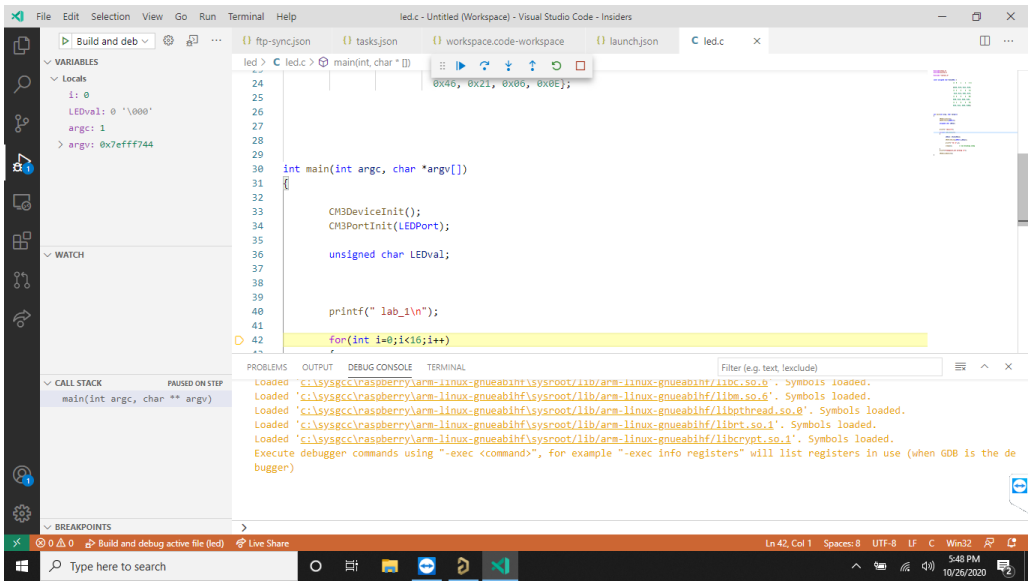


Figure 5 Example of a debugging session

7. Variables used by functions

Figure 6 Variables panel

1) The variables of the function will appear on the Variables panel on the side bar and are updated as the program runs. The local and global variable are shown. Using the Watch panel allows greater flexibility in displaying the variables.

2) Here in the picture the `LEDval` is still at zero, when the character which must be displayed '0' … 'F' will appear in this. This variable will increment in this panel.

8. Modular programming features

We have mentioned that the project management features of VS Code allow it to handle large projects. We will explore this feature now. The degree to which you wish to modularize a program is largely a matter of personal preference, but lab2.c is definitely too small for this! However, the introduction here will help later in the assignment.

9. Header files

1) Definitions, constants may distract. If there are a lot of them, it is better to put them into a separate header file.

2) First, select the statements to be put into the header file from lab1.c. ***Cut*** the following statement:

```
#define LEDPort 0x32
```

3) Then create a header file to keep the definitions. In the main menu, click on ***File*** / ***New File*** to create a new file. (This can also be done by clicking in the Activity bar, Explorer icon, hovering the mouse in the exploded *tree view* of LAB1 and the + icon for new file – you need to provide the file name `lab2.h` first).

4) In the empty editor screen of the new file, paste the text from the earlier step. The define statement should appear.

5) After all the required copying and pasting is done, save the file as `lab2.h`.

6) The statements in `lab2.h` need to be included back in `lab2.c` – to do so, select it and ***Type*** in `#include "lab2.h"` in the top of `lab2.c` (below `library.h`)

7) Build and test your program to verify that still works the same as before as if `LEDPort` is defined in the program.

10. Functions

1) In order to make a program easier to read, it is preferable to break it up into smaller, logical parts. This is so we can concentrate on the main task in the program and perhaps subdivide out the other parts for others to do. Let's put the conversion from binary to seven segment code into a routine.

2) As before create a new empty file ***Bin2LED.c***.

3) We want to transfer the entire Bin2LED array to `Bin2LED.c`

```
const unsigned char Bin2LED[]= ...
```

4) ***Cut*** the lines from `lab2.c` and ***Paste*** them into `Bin2LED.c`. Modify the function as in the appendix.
Also, modify the line:

```
LEDval = Bin2LED[i]; to LEDval = Bin2LED(i);
```

It is also good practice to put into `lab2.h`:

```
unsigned char Bin2LED(int);
```

Thinking question

In step 4, you changed the *array reference* to a *subroutine* by just changing square brackets [ ] to curved brackets (). Explain why this is so.

Appendix

Program template for Bin2LED.c:

```
unsigned char Bin2LED(int i)

{
      const unsigned char Bin2LED[] =

             /*  0     1     2    3 */
              {0x__, 0x__, 0x__, 0x__,
              /*  4     5     6    7*/
               0x__, 0x__, 0x__, 0x__,
              /*  8     9     A    B*/
               0x__, 0x__, 0x__, 0x__,
              /*  C     D     E    F*/
               0x__, 0x__, 0x__, 0x__};

      return(Bin2LED[i]);
}
```

Summary of keystrokes available in Visual Studio Code

Visual Studio Code

Keyboard shortcuts for Windows

General

Ctrl+Shift+P, F1	Show Command Palette
Ctrl+P	Quick Open, Go to File...
Ctrl+Shift+N	New window/instance
Ctrl+Shift+W	Close window/instance
Ctrl+,	User Settings
Ctrl+K Ctrl+S	Keyboard Shortcuts

Basic editing

Ctrl+X	Cut line (empty selection)
Ctrl+C	Copy line (empty selection)
Alt+ ↑ / ↓	Move line up/down
Shift+Alt + ↓ / ↑	Copy line up/down
Ctrl+Shift+K	Delete line
Ctrl+Enter	Insert line below
Ctrl+Shift+Enter	Insert line above
Ctrl+Shift+\	Jump to matching bracket
Ctrl+] / [	Indent/outdent line
Home / End	Go to beginning/end of line
Ctrl+Home	Go to beginning of file
Ctrl+End	Go to end of file
Ctrl+↑ / ↓	Scroll line up/down
Alt+PgUp / PgDn	Scroll page up/down
Ctrl+Shift+[	Fold (collapse) region
Ctrl+Shift+]	Unfold (uncollapse) region
Ctrl+K Ctrl+[	Fold (collapse) all subregions
Ctrl+K Ctrl+]	Unfold (uncollapse) all subregions
Ctrl+K Ctrl+0	Fold (collapse) all regions
Ctrl+K Ctrl+J	Unfold (uncollapse) all regions
Ctrl+K Ctrl+C	Add line comment
Ctrl+K Ctrl+U	Remove line comment
Ctrl+/	Toggle line comment
Shift+Alt+A	Toggle block comment
Alt+Z	Toggle word wrap

Navigation

Ctrl+T	Show all Symbols
Ctrl+G	Go to Line...
Ctrl+P	Go to File...
Ctrl+Shift+O	Go to Symbol...
Ctrl+Shift+M	Show Problems panel
F8	Go to next error or warning
Shift+F8	Go to previous error or warning
Ctrl+Shift+Tab	Navigate editor group history
Alt+ ← / →	Go back / forward
Ctrl+M	Toggle Tab moves focus

Search and replace

Ctrl+F	Find
Ctrl+H	Replace
F3 / Shift+F3	Find next/previous
Alt+Enter	Select all occurences of Find match
Ctrl+D	Add selection to next Find match
Ctrl+K Ctrl+D	Move last selection to next Find match
Alt+C / R / W	Toggle case-sensitive / regex / whole word

Multi-cursor and selection

Alt+Click	Insert cursor
Ctrl+Alt+ ↑ / ↓	Insert cursor above / below
Ctrl+U	Undo last cursor operation
Shift+Alt+I	Insert cursor at end of each line selected
Ctrl+L	Select current line
Ctrl+Shift+L	Select all occurrences of current selection
Ctrl+F2	Select all occurrences of current word
Shift+Alt+→	Expand selection
Shift+Alt+←	Shrink selection
Shift+Alt + (drag mouse)	Column (box) selection
Ctrl+Shift+Alt + (arrow key)	Column (box) selection
Ctrl+Shift+Alt +PgUp/PgDn	Column (box) selection page up/down

Rich languages editing

Ctrl+Space	Trigger suggestion
Ctrl+Shift+Space	Trigger parameter hints
Shift+Alt+F	Format document
Ctrl+K Ctrl+F	Format selection
F12	Go to Definition
Alt+F12	Peek Definition
Ctrl+K F12	Open Definition to the side
Ctrl+.	Quick Fix
Shift+F12	Show References
F2	Rename Symbol
Ctrl+K Ctrl+X	Trim trailing whitespace
Ctrl+K M	Change file language

Editor management

Ctrl+F4, Ctrl+W	Close editor
Ctrl+K F	Close folder
Ctrl+\	Split editor
Ctrl+ 1 / 2 / 3	Focus into 1st, 2nd or 3rd editor group
Ctrl+K Ctrl+ ←/→	Focus into previous/next editor group
Ctrl+Shift+PgUp / PgDn	Move editor left/right
Ctrl+K ← / →	Move active editor group

File management

Ctrl+N	New File
Ctrl+O	Open File...
Ctrl+S	Save
Ctrl+Shift+S	Save As...
Ctrl+K S	Save All
Ctrl+F4	Close
Ctrl+K Ctrl+W	Close All
Ctrl+Shift+T	Reopen closed editor
Ctrl+K Enter	Keep preview mode editor open
Ctrl+Tab	Open next
Ctrl+Shift+Tab	Open previous
Ctrl+K P	Copy path of active file
Ctrl+K R	Reveal active file in Explorer
Ctrl+K O	Show active file in new window/instance

Display

F11	Toggle full screen
Shift+Alt+0	Toggle editor layout (horizontal/vertical)
Ctrl+ = / -	Zoom in/out
Ctrl+B	Toggle Sidebar visibility
Ctrl+Shift+E	Show Explorer / Toggle focus
Ctrl+Shift+F	Show Search
Ctrl+Shift+G	Show Source Control
Ctrl+Shift+D	Show Debug
Ctrl+Shift+X	Show Extensions
Ctrl+Shift+H	Replace in files
Ctrl+Shift+J	Toggle Search details
Ctrl+Shift+U	Show Output panel
Ctrl+Shift+V	Open Markdown preview
Ctrl+K V	Open Markdown preview to the side
Ctrl+K Z	Zen Mode (Esc Esc to exit)

Debug

F9	Toggle breakpoint
F5	Start/Continue
Shift+F5	Stop
F11 / Shift+F11	Step into/out
F10	Step over
Ctrl+K Ctrl+I	Show hover

Integrated terminal

Ctrl+`	Show integrated terminal
Ctrl+Shift+`	Create new terminal
Ctrl+C	Copy selection
Ctrl+V	Paste into active terminal
Ctrl+↑ / ↓	Scroll up/down
Shift+PgUp / PgDn	Scroll page up/down
Ctrl+Home / End	Scroll to top/bottom

Other operating systems' keyboard shortcuts and additional unassigned shortcuts available at aka.ms/vscodekeybindings

Laboratory 3 Keypad Interfacing

1. Introduction

In this lab, you will use the buffers and latches of the I/O board to scan a keypad, and output the key pressed to an LED. You will also have to map the scanned key result to the actual value on the keypad.

2. Objectives

- To perform keypad scanning on a 4x3 key device
- Learn about interfacing to a keypad

3. Interfacing to a keypad

1) Telephone keypads are an economical, commonly available input device. They are made up of wires arranged as a matrix. A pair of wires will make contact when a key is pressed. For convenience, these wires end up as connector pins on the keypad. The sample which we are using has 8 pins and 12 keys.

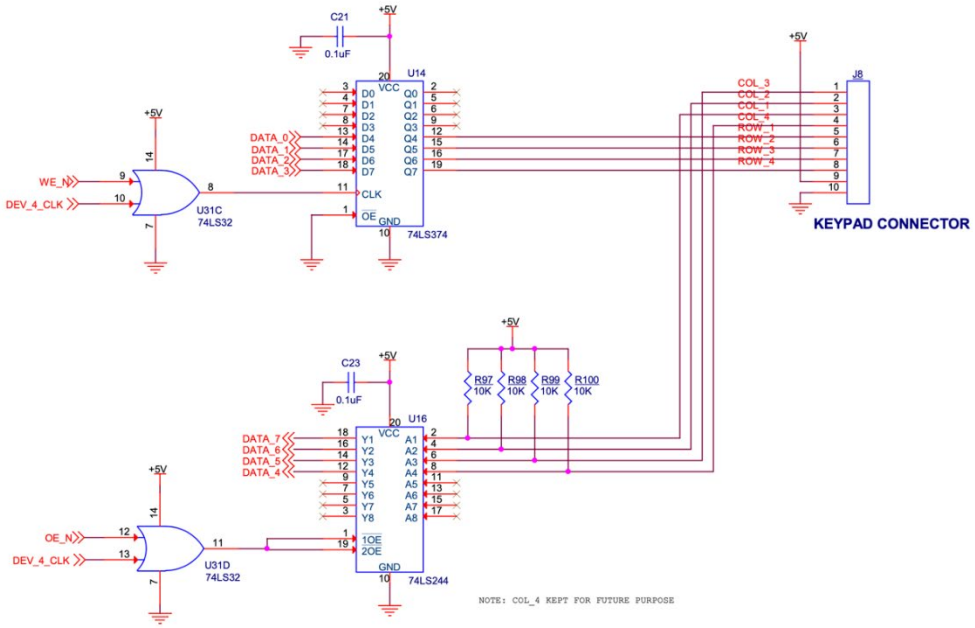
2) To interface a keypad in practice, we need to do the following:

   i) Determine which pins make contact when a key is pressed
   ii) Construct a connector to join the keypad to the processor board
   iii) Specify which bits in the processor match the connector pins.

   This lab will go through these design steps.

3) In the I/O Board, we will be using Device 4 to do keypad detection. This device is configured so that the upper nybble is configured as an input and the lower nybble is configured as an output. This is shown in the following figure.

4) To save some effort, the key connection table is given below. Looking at the keypad face up, and taking the leftmost pin as 1, the following pins short when the corresponding keys are pressed. For example, when you press the '3' key, pins 3 and 4 will make contact.



Keypad	pin 4	pin 5	pin 6	pin 7
pin 3	3	6	9	#
pin 2	2	5	8	0
pin 1	1	4	7	*

Figure 1: Key connection table

5) How do we detect a keypress? We may easily do so by using an 8 bit bidirectional port, or at least one with 3 output and 4 input pins. A common way will be to pull up pins 1 to 3 of the keypad with resistors, and output a '0' from pins 4 to 7 one at a time. Then pins 1 to 3 are checked for the presence of a '0'.

6) Let's say we have output a '0' to pin 4. If the '3' key was pressed, we would see a '0' at pin 3 when we read in pins 1 to 3.

7) Connector pins 1 to 3 are *input* from pins 3 to 1 on the keypad, and connector pins 5 to 8 are *output* to pins 4 to 7 on the keypad. Remember, if NO keys were pressed, pins 1 to 3 on the keypad will return all 1's.

8) The connector joins these pins to the processor board. It has its own numbering. But it will map to the processor port and this is the portion that is of interest.
Thus, from the key connection table which is given, we should be able to determine the corresponding port bits of interest.

9) You may experience some keyboard bounce problems, which may be overcome by detecting a key press, and reading the same key again after a delay to confirm that it is a valid keypress.

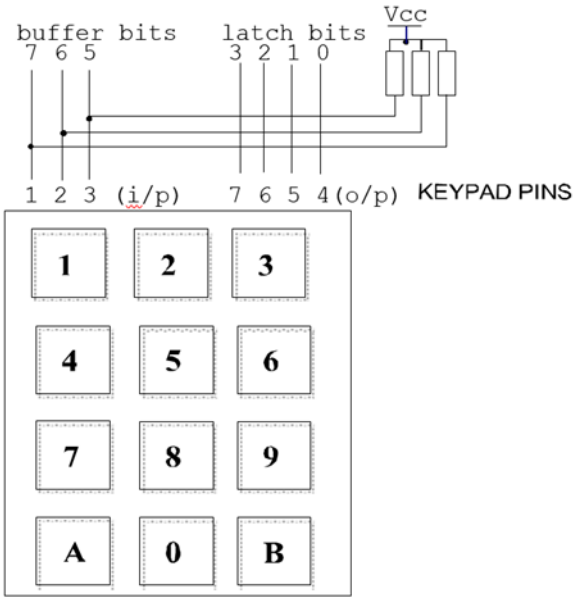


Figure 2: Schematic of Keypad connection

4. Program listing notes

1) The main program runs as a continuous loop, calling the function `ScanKey()` which performs one round of scanning the four columns. It returns `0xFF` if no key is pressed and thus `ScanKey()` can be easily used in other routines. If a key is pressed, it is converted to the equivalent 7 segment code by means of an array, so it can be displayed.

2) The program assumes the existence of a port that has the higher nybble as an output and the lower nybble as an input. In many microcontrollers the ports, are bidirectional. Here, we use a latch and buffer respectively, where some bits are not used.

3) A “walking zero” bit pattern as described in the lecture, will be output to the latch. This bit pattern will be the higher nybble of a byte. If a key is pressed, the buffer will have a ‘0' voltage level read into one of its four bit positions. Combining the input pattern with the output pattern, we will have a unique combination of bits that will identify which key is pressed. This is called the *scan code*.

4) The routine `ProcKey()` will search the `ScanTable` sequentially to find the scan code. That is, `ScanTable` is arranged so that the first entry will correspond to the ‘0' key, the second to the ‘1' key and so on. So by counting at which position in `ScanTable` the *detected scancode* is, we know what key is pressed.

5. Instructions

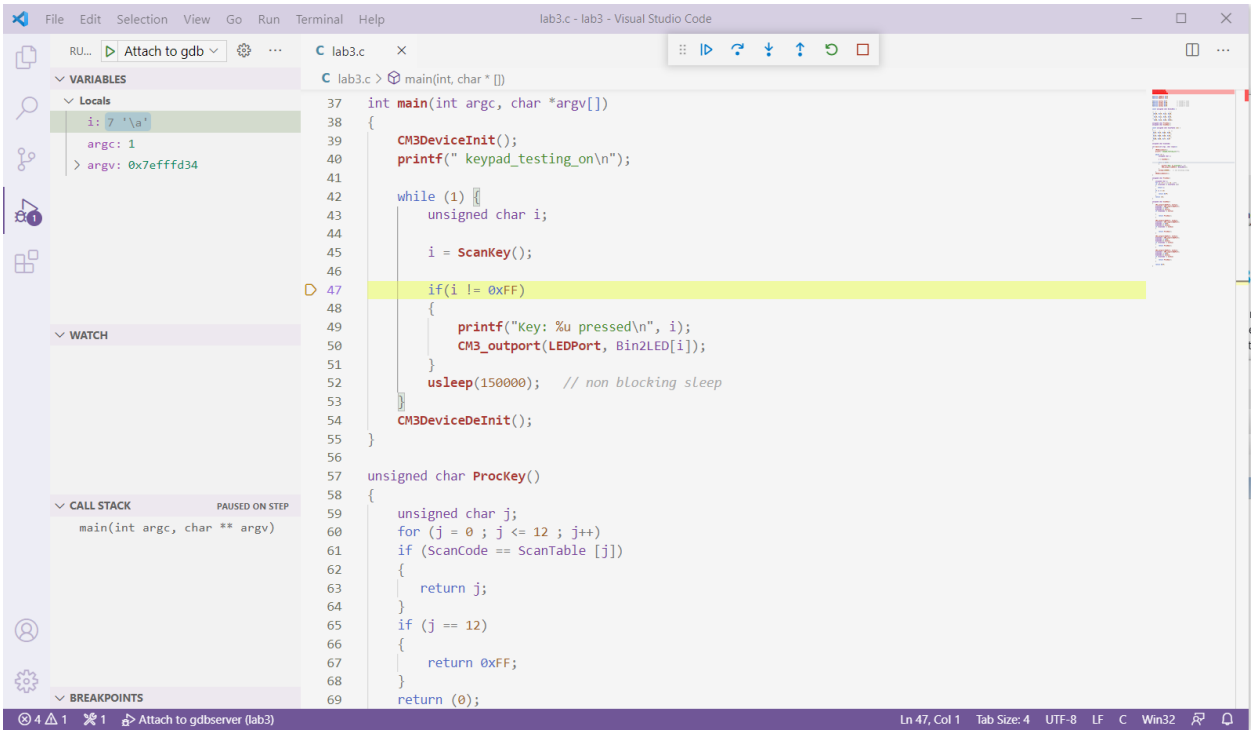
1) As in the previous labs, the `ECSLAB\lab3` directory has been created. The file `lab3.c` with the blanks as shown below, is given to you.

2) Refer to `lab3.c` in the appendix and fill in the blanks, with reference to the values generated by the various key presses. Use the schematic of the key pad as shown above. Compile the program and load it into the CM3.

Appendix – Hints to debug a keypress and program template for lab3

Using VS Code to debug a keypress

In VS Code, the return value of the `ScanKey` function corresponds to key pressed when executing the function. If the key '7' is pressed, you will notice the value `'\a'` appear in the variable `i` in the debug pane. This is actually the ASCII representation of the value 0x07. You may inspect other keypad return values during the lab.



Program template for lab 3

```
#include<stdio.h>
#include<unistd.h>
#include "library.h"

#define LEDPort 0x32
#define KbdPort 0x34

#define Col7Lo 0x__            // column 7 scan
#define Col6Lo 0x__            // column 6 scan
#define Col5Lo 0x__            // column 5 scan
#define Col4Lo 0x__            // column 4 scan

const unsigned char Bin2LED[] =
```

```
/*  0     1     2    3 */
 {0x__, 0x__, 0x__, 0x__,
/*  4     5     6    7*/
  0x__, 0x__, 0x__, 0x__,
/*  8     9     A    B*/
  0x__, 0x__, 0x__, 0x__};

unsigned char ProcKey();
unsigned char ScanKey();

const unsigned char ScanTable [12] =
{
/* 0     1     2     3 */
 0x__, 0x__, 0x__, 0x__,
/* 4     5     6     7 */
 0x__, 0x__, 0x__, 0x__,
/* 8     9     *     # */
 0x__, 0x__, 0x__, 0x__
};

unsigned char ScanCode;

int main(int argc, char *argv[])
{
      CM3DeviceInit();
      printf(" keypad_testing_on\n");
      while (1) {
            unsigned char i;
            i = ScanKey();

            if(i != 0xFF){
                  printf("Key: %u pressed\n", i);
                  CM3_outport(LEDPort, Bin2LED[i]);
            }
            usleep(150000);    // non blocking sleep
      }
      CM3DeviceDeInit();
}
unsigned char ProcKey()
{
      unsigned char j;
      for (j = 0 ; j <= 12 ; j++)
      if (ScanCode == ScanTable [j])
      {
         return j;
      }
      if (j == 12)
```

```
            {
                  return 0xFF;
            }
            return (0);
      }
unsigned char ScanKey()
{
      CM3_outport(KbdPort, Col7Lo);
      ScanCode = CM3_inport(KbdPort);
      ScanCode |= 0x0F;
      ScanCode &= Col7Lo;
      if (ScanCode != Col7Lo)
      {
          return ProcKey();
      }
      CM3_outport(KbdPort, Col6Lo);
      ScanCode = CM3_inport(KbdPort);
      ScanCode |= 0x0F;
      ScanCode &= Col6Lo;
      if (ScanCode != Col6Lo)
      {
          return ProcKey();
      }

      CM3_outport(KbdPort, Col5Lo);
      ScanCode = CM3_inport(KbdPort);
      ScanCode |= 0x0F;
      ScanCode &= Col5Lo;
      if (ScanCode != Col5Lo)
      {
          return ProcKey();
      }
      CM3_outport(KbdPort, Col4Lo);
      ScanCode = CM3_inport(KbdPort);
      ScanCode |= 0x0F;
      ScanCode &= Col4Lo;
      if (ScanCode != Col4Lo)
      {
          return ProcKey();
      }
      return 0xFF;
}
```

SINGAPORE POLYTECHNIC
SCHOOL OF ELECTRICAL AND ELECTRONICS ENGINEERING

ET0104 Embedded Computer Systems Laboratory

Laboratory 4 Interfacing to a Liquid Crystal Display

1. Introduction

In this lab, you will use the CM3 I/O Board to scan a keypad and interact with a LCD module.

2. Objectives

- To interface to a Liquid Crystal Display module (LCD)
- To perform keypad scanning on a 4x3 device
- How to use combined LCD, LED, Keypad using GPIO.

3. Interfacing to an LCD module

1) Liquid Crystal Displays (LCD) are a popular display device. More commonly LCDs come in the form of modules. These incorporate controller chips with driver functions on board. Some of its advantages are low power, ease of programming and low overhead in terms of refreshing such a device. To make more efficient use of these modules, some points must be noted.

2) Firstly, a 4 bit data transfer protocol is preferred, as this uses up less pins.

3) Secondly, the ENABLE pin cannot be driven by the hardware generated control signals of the SBC such as the R/-W or E pins. A software generated signal has to be used. In our case this means that to toggle a bit, we set the bit to low, write to a latch, set the bit to high, write again, and write one more time with the bit low.

4) Third, certain operations take up more time than others. It is possible to read the status of the LCD, but that may involve extra hardware and decoding. A standard approach is to assume a common time whereby all operations are complete. Note that since different LCD modules may exhibit different execution times, it may be worthwhile to adjust the time delay accordingly.

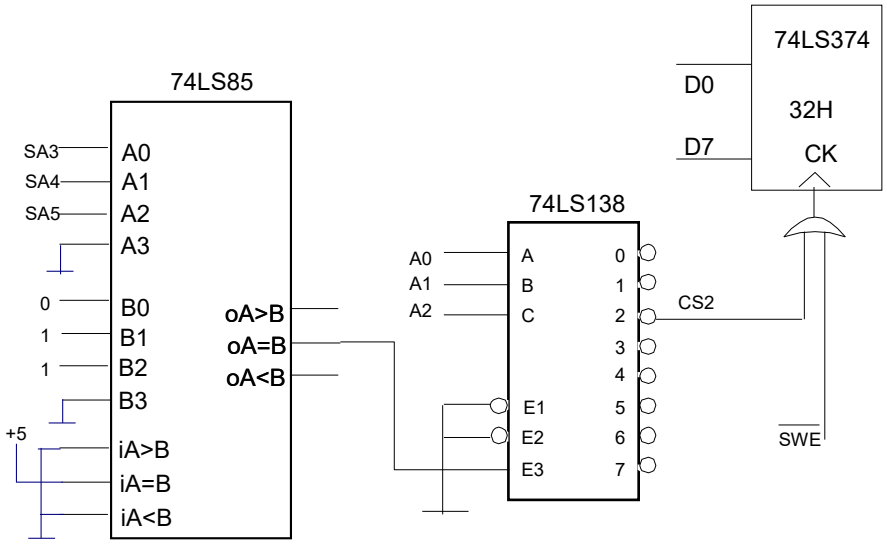
5) We will use the same keypad from the previous laboratory, but this time, we will output to the LCD module. The following gives the layout of the pins.

Pin layout of LCD latch:	7	6	5	4	3	2	1	0
LCD Control/Data pins	D7	D6	D5	D4	X	E	R/W	RS

6) I/O Addresses

The SMI bus of the CM3 allows a maximum of 64 I/O devices to be accessed. It is useful to have the flexibility of setting these addresses to fall within a certain range. The advantage of the 74LS85 is that it allows the fixing of this range by just setting a few DIP switches.

In this lab we want the LED which is set up as Device 2, to be at address 2AH. The hardware circuit is shown below.



Fill in the corresponding values for the LCD and KeyPad, which are Devices 3 and 4 respectively. You can find the required information from earlier labs about the I/O map for the board. Set the DIP switches accordingly. The following truth table is given to help you:

Device/Addr	A5	A4	A3	A2	A1	A0
LED						
LCD						
KeyPad						

4. Instructions

1) The program lab4.c is given in the appendix. Fill in the blanks, using the LCD instruction sheet if necessary. Load it into the development system and test it to see if a message appears and what is typed into the keypad is reflected in the display.

2) Using `sprintf` for more versatile displays.

i) We will modify `lab4.c` now to try out some useful printing functions. The array `LCDStr` accepts a pointer to a C-formatted text string. That is, the end of the string is indicated by a 0x00. We can use the C function `sprintf` to generate message strings for us.The advantages are many; for example, data conversions from all kinds of number formats to text strings are done for us. However, we have to use the C formatting commands as found in the `printf` command. We also have great flexibility in composing strings using functions like `strcat` (concatenate strings) and so on.

ii) **Modify** lab4.c as follows:

1. Declare new variables as follows:

```
char LCDStr[17];     /* 16 char LCD + 1 for end of string */
char test;           /* to test */
```

2. In the main program, *replace* the line `LCDprint("12345678");` by:

```
test=104;
sprintf(LCDStr,"Subject:ET%d-OK”,test); /* don’t exceed num chars!*/
LCDprint(LCDStr);
```

What do you expect to see?

3. Another way of doing the above, illustrating the use of `strcat`.

```
test=104;sprintf(LCDStr,"Subject:ET%d",test); /*generate 1st string */
strcat(LCDStr,"-OK");                         /* append using strcat */
LCDprint(LCDStr);
```

4. Modify point 3. above by displaying a floating point conversion with the use of a message string kept elsewhere – that is *not* part of the `sprintf` statement. For example, display the price of an item ($104) with GST computed. (Hint: do an online search for “format specifiers in C”).

Appendix

Program template for lab 4:

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include "library.h"

#define LEDPort ____
#define KbdPort ____
#define LCDPort ____

unsigned char ProcKey();
unsigned char ScanKey();

const unsigned char ScanTable [12] =
{
/* 0     1     2     3 */
0x__, 0x__, 0x__, 0x__,
/* 4     5     6     7 */
0x__, 0x__, 0x__, 0x__,
/* 8     9     *     # */
0x__, 0x__, 0x__, 0x__
};

const unsigned char Bin2LED[] =

      /*  0     1     2    3 */
      {0x__, 0x__, 0x__, 0x__,

      /*  4     5     6    7*/
      0x__, 0x__, 0x__, 0x__,

      /*  8     9     A    B*/
      0x__, 0x__, 0x__, 0x__};

unsigned char ScanCode;

#define Col7Lo ____            // column 7 scan
#define Col6Lo ____            // column 6 scan
#define Col5Lo ____            // column 5 scan
#define Col4Lo ____            // column 4 scan

static void initlcd();
static void lcd_writecmd(char cmd);
static void LCDprint(char *sptr);
static void lcddata(unsigned char cmd);
```

```
int main(int argc, char *argv[])
{
CM3DeviceInit();

initlcd();
lcd_writecmd(0x80);
LCDprint("LCD Lab");
lcd_writecmd(0xC0);
LCDprint("12345678");

while(1)
{
      unsigned char i,ii;
      i = ScanKey();
      if (i != 0xFF)
      {
            if (i > 0x39) {
                  ii = i - 0x37;
            } else {
                  ii = i - 0x30;
            }
            lcddata(i);
            CM3_outport(LEDPort, Bin2LED[ii]);
            usleep(300000);
      }
}
CM3DeviceDeInit();
}

static void initlcd(void)
{
       usleep(20000);
       lcd_writecmd(0x30);
       usleep(20000);
       lcd_writecmd(0x30);
       usleep(20000);
       lcd_writecmd(0x30);

       lcd_writecmd(0x__);  // 4 bit mode
       lcd_writecmd(0x__);  // 2 line  5*7 dots
       lcd_writecmd(0x__);  //clear screen
       lcd_writecmd(0x__);  //dis on cur off
       lcd_writecmd(0x__);  //inc cur
lcd_writecmd(0x80);
}
static void lcd_writecmd(char cmd)
{
```

```
char data;

data = (cmd & 0xf0);
CM3_outport(LCDPort, data | 0x04);
usleep(10);
CM3_outport(LCDPort, data);

usleep(200);

data = (cmd & 0x0f) << 4;
CM3_outport(LCDPort, data | 0x04);
usleep(10);
CM3_outport(LCDPort, data);

usleep(2000);
}
static void LCDprint(char *sptr)
{
while (*sptr != 0)
{
      int i=1;
      lcddata(*sptr);
      ++sptr;
}
}
static void lcddata(unsigned char cmd)
{
char data;

data = (cmd & 0xf0);
CM3_outport(LCDPort, data | 0x05);
usleep(10);
CM3_outport(LCDPort, data);
usleep(200);

data = (cmd & 0x0f) << 4;
CM3_outport(LCDPort, data | 0x05);
usleep(10);
CM3_outport(LCDPort, data);

usleep(2000);
}
/*----------- Keypad Functions --------------*/

unsigned char ScanKey()
{
CM3_outport(KbdPort, Col7Lo);
```

```
    ScanCode = CM3_inport(KbdPort);
    ScanCode |= 0x0F;
    ScanCode &= Col7Lo;
    if (ScanCode != Col7Lo)
    {
        return ProcKey();
    }
    CM3_outport(KbdPort, Col6Lo);
    ScanCode = CM3_inport(KbdPort);
    ScanCode |= 0x0F;
    ScanCode &= Col6Lo;
    if (ScanCode != Col6Lo)
    {
        return ProcKey();
    }
    CM3_outport(KbdPort, Col5Lo);
    ScanCode = CM3_inport(KbdPort);
    ScanCode |= 0x0F;
    ScanCode &= Col5Lo;
    if (ScanCode != Col5Lo)
    {
        return ProcKey();
    }
    CM3_outport(KbdPort, Col4Lo);
    ScanCode = CM3_inport(KbdPort);
    ScanCode |= 0x0F;
    ScanCode &= Col4Lo;
    if (ScanCode != Col4Lo)
        return ProcKey();
    }
    return 0xFF;
    }
unsigned char ProcKey()
{
    unsigned char j;
    for (j = 0 ; j <= 12 ; j++)
    if (ScanCode == ScanTable [j])
    {
       if(j > 9) {
             j = j + 0x37;
       } else {
             j = j + 0x30;
       }
       return j;
    }
    if (j == 12)
    {
```

```
            return 0xFF;
        }
        return (0);
    }
```

Instruction	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution Time
Clear Display	0	0	0	0	0	0	0	0	0	1	Clears Display and returns cursor to the Home Position (Address 00)	80uS = 1.64mS
Return Home	0	0	0	0	0	0	0	0	1	*	Returns cursor to Home Position. Returns shifted display to original position. Does not clear display	40uS = 1.6mS
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S	Sets DD RAM counter to increment or decrement (I/D) Specifies cursor or display shift during to Data Read or Write (S)	40uS
Display ON/OFF Control	0	0	0	0	0	0	1	D	C	B	Sets Display ON/OFF (D), cursor ON/OFF (C), and blink character at cursor position	40uS
Cursor or Display Shift	0	0	0	0	0	1	S/C	R/L	*	*	Moves cursor or shifts the display w/o changing DD RAM contents	40uS
Function Set	0	0	0	0	1	DL	N	F	*	*	Sets data bus length (DL), # of display lines (N), and character font (F)	40uS
Set CG RAM Address	0	0	0	1	A_{CG}						Sets CG RAM address. CG RAM data is sent and received after this instruction	40uS
Set DD RAM Address	0	0	1	A_{DD}							Sets DD RAM address. DD RAM data is sent and received after this instruction	40uS
Read Busy Flag & Address	0	1	BF	AC							Reads Busy Flag (BF) and address counter contents	1uS
SIZE=2>Write Data from DD or CG RAM	1	0	Write Data								Writes data to DD or CG RAM and increments or decrements address counter (AC)	40uS
Read Data from DD or CGRAM	1	1	Read Data								Reads data from DD or CG RAM and increments or decrements address counter (AC)	40uS
I/D=1: Increment S=1: Display Shift on data entry S/C=1: Display Shift (RAM unchanged) R/L=1: Shift to the Right DL=1: 8 bits N=1: 2 Lines F=1: 5x10 Dot Font D=1: Display ON C=1: Cursor ON B=1: Blink ON BF=1: Cannot accept instruction				I/D=0: Decrements S=0: Cursor Shift on data entry S/C=0: Cursor Shift (RAM unchanged) R/L=0: Shift to the Left DL=0: 4 bits N=0: 1 Line F=0: 5x7 Dot Font D=0: Display OFF C=0: Cursor OFF B=0: Blink OFF BF=0: Can accept instruction							Definitions: DD RAM: Display data RAM CG RAM: Character generator RAM A_{CG}: CG RAM Address A_{DD}: DD RAM Address(Cursor Address) AC: Address Counter used for both DD and CG RAM Address	Execution Time changes when Frequency changes per the following example: If F_{CP} or f_{osc} is 27 KHz 40uS x 250/270 = 37uS

SINGAPORE POLYTECHNIC
SCHOOL OF ELECTRICAL AND ELECTRONICS ENGINEERING

ET0104 Embedded Computer Systems Laboratory

Laboratory 5 - Controlling a stepper motor

1. Introduction

Stepper motors are commonly used to provide motion control. Their popularity comes from their ease of use. To move it one step you send one pulse from a processor. To drive a load at a given speed, all that is needed is a properly timed set of pulses. The upper speed limit is controlled by its torque speed characteristics.

Stepper motors are rated by:

i) The angle the motor shaft rotates when one pulse is sent
ii) The current it draws - which is a measure of the load it can drive

2. Objectives

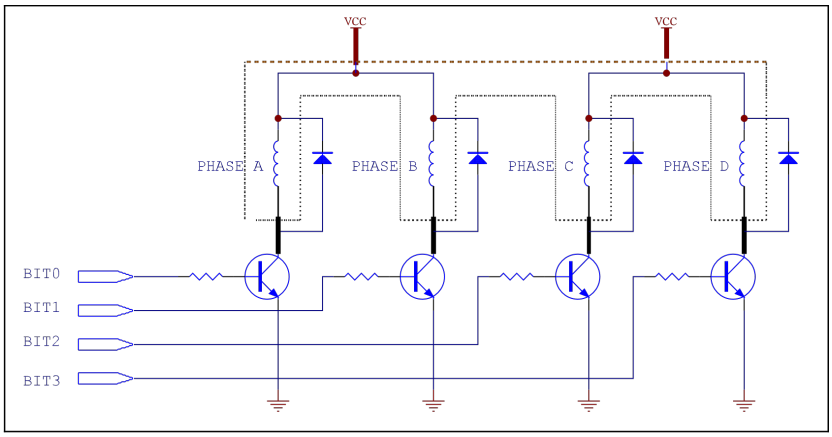
- To experiment with various types of stepper motor movements

- To move a stepper motor continuously and with acceleration

3. Interfacing to a stepper motor

1) We can drive the motor with one phase, two phase, or one-two phase on. We use only the lower four bits of a latch to turn the transistor on or off.

2) In the following diagram, the internal electrical connections of the motor itself is represented by the dotted lines. It is connected to the I/O board via a 6 pin connector which is plugged into the STPMTR connector located at the lower left of the board.



Schematic of a four phase stepper motor circuit

4. Positioning Applications

A table below is given for you to fill. The processor will fetch one byte at a time from this table and output it to the stepper motor latch. This will turn on the matching transistor as shown in the earlier diagram. We are considering three types of drive schemes. For each of them, we will subjectively compare the torque the motor produces. To do so, hold the base of the motor mount and *lightly* hold the motor shaft to stop it from rotating, but *do not* force the motor!

5. Full step - one phase on

1) The first drive scheme we will consider is the full step one phase on motion. This means we want to make the motor move in full steps, one phase on at a time. A '1' turns the phase on.

Step	Phase D	Phase C	Phase B	Phase A	Value
1	0	0	0	1	00000001
2					
3					
4					
1					

TABLE 1 Full-Step One phase on sequence for clockwise rotation.

2) Complete the table and enter the values into LAB5.C at the section labelled `PTable`. See the appendix. There can be 4 or 8 entries, depending on the type of move.

For this application, we want to rotate the motor 360 degrees. The motor given is rated to rotate 1.8 degrees per step. The variable `NumSteps` in the program will control the number of steps the motor rotates. Calculate the value for one complete rotation.

NumSteps _____

Assemble the program, load and run it. See if the motor rotates one revolution. *Gently* feel the motor shaft as mentioned earlier, to get a feel of the torque generated.

6. Full step- one phase on-reverse

Now we want to make the motor rotate in the opposite direction for one revolution, one phase on at a time. Compute the values needed and modify `lab5.c`.

Step	Phase D	Phase C	Phase B	Phase A	Value
1	1	0	0	0	00001000
2					
3					
4					

TABLE 2 Full-Step two phase on sequence for anti-clockwise rotation.

7. Full step- two phase on

Now we want to make the motor move full steps, two phases on at a time in the forward direction. Again, compute the values and modify motor.c

Step	Phase D	Phase C	Phase B	Phase A	Value
1	0	0	1	1	00000011
2					
3					
4					

TABLE 3 Full-Step two phase on sequence for clockwise rotation.

Gently feel the motor shaft as mentioned earlier, to get a feel of the torque generated - the two phase on drive should have more torque.

8. Half step

Finally, we make the motor move in half steps. Complete the table and proceed as before.

Step	Phase D	Phase C	Phase B	Phase A	Value
1					
2					
3					
4					
5					
6					
7					
8					

TABLE 4 Half-Step Sequence for clockwise rotation.

9. Instructions for positioning a motor

Run the program again, and *lightly* touch the motor shaft. Does it seem to have more vibration as compared to the previous movements?

10. Speed applications

To move at a given speed, we need to introduce a time delay in between steps. While a motor moves in discrete steps, motor speed is normally given in terms of revolutions per minute (rpm). Thus we have to perform some conversions.

In this part of the lab, we want the motor to act as the "second" hand of a clock. That is, it should turn one revolution in one minute. Again, one step takes 1.8 degrees. Complete the following calculations

Number of steps sent per minute _____

Time duration between each step _____

Make all these changes to lab5.c. Load and test it. Do you have an accurate "clock"?

Appendix

Program template for lab5:

```
/*********************************/
/*      Stepper Motor lab        */
/*********************************/

#include <stdio.h>
#include <unistd.h>
#include "library.h"

#define SMPort 0x31    /* stepper motor port */
#define NumSteps 200   /* number of steps to move */
#define PtableLen4     /* number of entries in Phase table */

unsigned char Ptable []={0x__,0x__,0x__,0x__};

/********** MAIN PROGRAM *************/

int main(int args, char *argv[])
{
      int i,j;
      CM3DeviceInit();

      printf("starting_stepper_motor\n");
      i=0;
      for (j=NumSteps;j>0;j--)
      {
            CM3_outport(SMPort, Ptable[i]);    /* output to stepper motor */
            usleep(10000);                     /* delay */
            i++;                               /* pointer to phase table */
            if (i>=PtableLen) i=0;             /* recycle at end of table */
      }
      CM3DeviceDeInit();
}
```

SINGAPORE POLYTECHNIC
SCHOOL OF ELECTRICAL AND ELECTRONICS ENGINEERING

ET0104 Embedded Computer Systems Laboratory

Laboratory 6 - Digital to Analogue, Analogue to Digital Interfacing

1. Introduction

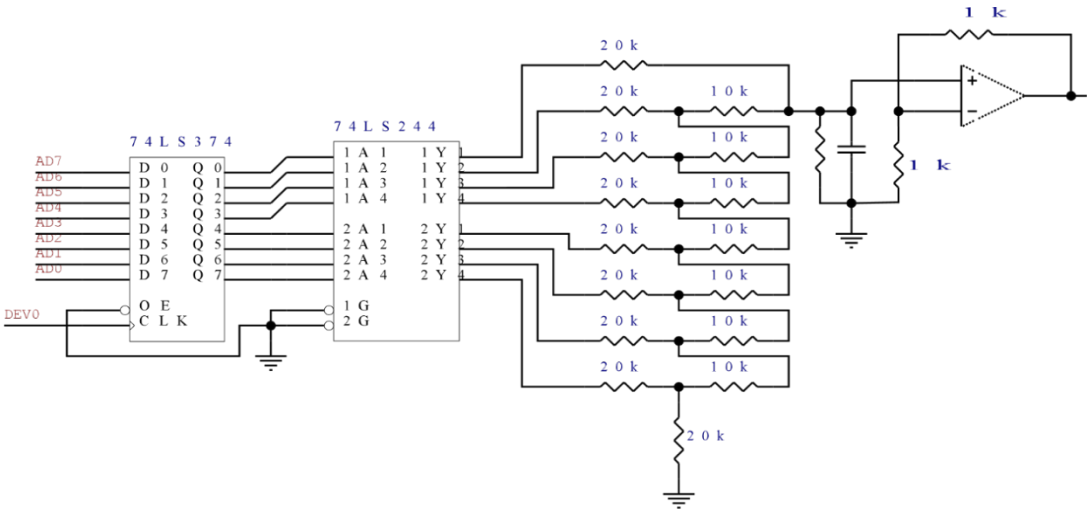
In this lab, you will use the SBC to output various values to an R-2R network.

2. Objectives

- To configure a latch and buffer to work with a R-2R network
- To anticipate the output of a D/A converter given various digital inputs
- To observe the working of a A/D converter

3. Digital to Analogue conversion

1) Digital to Analogue (D/A) interfacing allows a microcontroller to perform analogue control, as opposed to on-off control for purely digital systems. Most D/A converters are made of integrated circuits. These circuits are made up of R-2R networks anyway, with various degrees of quality in the integrated components. In this lab we are using a voltage mode converter. Note the use of the 74LS244 which increases the current drive of the 74LS373 latch. The opamp acts as a buffer and provides a gain of 2.



R-2R ladder in voltage conversion mode

2) First we need to find out the resolution of the D/A converter. This will help us in various calculations later. The resolution is the smallest change in the analogue output, for the smallest change in the digital input. The following program lab6.C will generate a certain waveform: what is its shape?

```
#define DACPort 0x30
unsigned char DACout;

void main()
{
DACout = 0;
while TRUE
    {
        CM3_output(DACPort, DACout);    /* DAC Port */
        DACout++;                       /* increment */
     }
}
```

3) To answer this, consider the smallest and largest values DACout will take.

4. Initial observations

Now power up the I/O Board, and load the program lab6.C. Run the program and this time, power on the oscilloscope and attach a probe at the connector DA_OUT, located near the top left of the board.

What is the maximum *digital* value that will be output to the R-2R circuit? _____

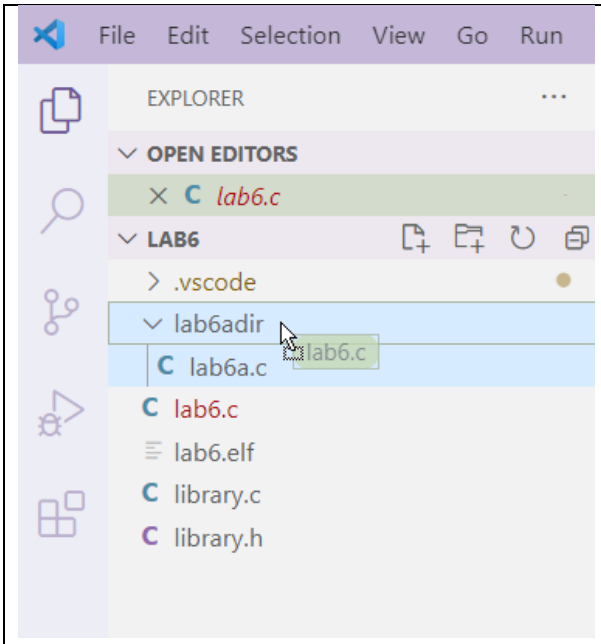
What is the maximum *analog* value of the wave you see? _____

What is the resolution of the D/A circuit? _____

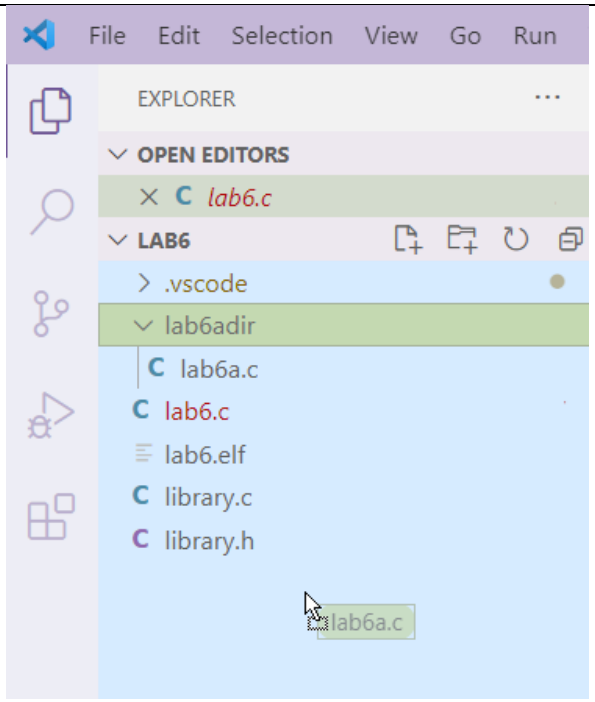
Note that if the waveform was distorted, we should only use the linear portion.

5. Generating a sine wave

We want to bring in another program to generate a sine wave. In our setup, we cannot have two C programs having a `main` function in the same directory. For convenience, we have kept this program in a separate sub-directory called lab6a in lab6. We have to *move* lab6.c into the sub-directory lab6a and *move* the file lab6a.c to the directory lab6.



i) ***Drag*** the file lab6.c to *folder* lab6adir. Click on the ***Move*** button when prompted.



ii) ***Drag*** the file lab6a.c from *folder* lab6adir to *folder* lab6 – bring the mouse cursor past the last file in the folder. Click on the ***Move*** button when prompted.

Check to make sure `lab6a.c` is in *folder* lab6 and `lab6.c` is in the *subfolder* lab6adir.

1) Using the program LAB6A.C, we put in data and count values so we can see a sine wave at the DAC output.

We note that:

i) This hardware configuration cannot output a negative voltage. If we need to generate a sine wave, we need to add an offset to it.

ii) In order to minimize the quantization error, we want the maximum value of the waveform to be reached when the maximum digital value is output.

2) In general, the equation of a sine wave with offset is:

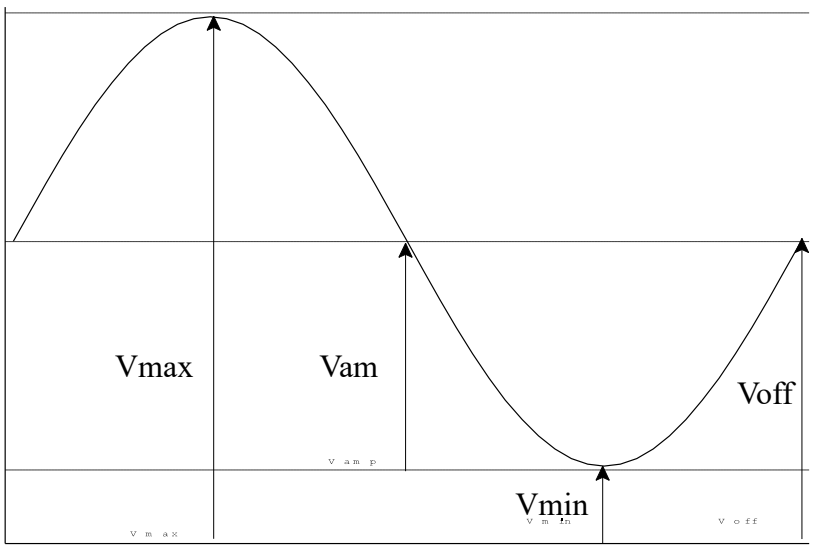
$$Vout = Voff + Vamp * \sin \theta$$

3) From the diagram, the value of the offset voltage,

$$Voff = (Vmax + Vmin) / 2$$

4) Hence the amplitude,

$$Vamp = (Vmax - Vmin) / 2$$



General equation of a sine wave with offset

5) For our lab, we let the minimum be zero, so the sine wave is:

$$V = (Vamp * \sin\theta) + Voff \quad \text{or; } Vmax/2\ (1+ \sin\theta)$$

6) We have seen that the resolution 1 / ρ, is the voltage represented by one bit for the DAC. The *scale factor* F_{scale} is the digital value for one volt and is the reciprocal of the resolution, 1 / ρ.

7) From the previous measurement, the value of F_{scale} is: _____

8) The table below will assist you in the calculation of the necessary values for the generation of a sine wave using 12 equal intervals

θ	0	30	60	90	120	150	180	210	240	270	300	330
$\sin\theta$	0	0.5	0.87	1	0.87	0.5	0	-0.5	-0.87	-1	-0.87	-0.5
$V=(V_{max}/2)*(1+\sin\theta)$												
$F_{scale} * V$												

6. Instructions

1) Substitute the calculated values into the appropriate data locations in LAB6B.C. Execute the program and observe the output on the oscilloscope.

2) What would we have to change in order to obtain

   i) a smoother sine wave?

   _____

   ii) a higher/lower frequency waveform?

   _____

7. Optional Exercise

1) Generate a half wave rectified sine signal, display and show your lecturer.

Laboratory 7 - Graphics Display Technology

1. Introduction

Modern embedded systems have powerful graphics display capabilities. For example, the CM3 processor has a dedicated Graphics Processing Unit (GPU) included as part of its SoC (System on Chip). This allows it to render graphics quickly without much load to the main CPU. Various graphics libraries are available to display images in high resolutions, in our case through a HDMI port.

2. Objectives

- Setting up VS Code to generate an application that uses graphics and a local filesystem
- Prepare images that can be displayed on a target system
- To transfer images to an embedded system.

3. Simple display application and file transfer

To demonstrate the capabilities of the system, we first show a simple application which displays images through some simple user interaction. To do so, we will transfer some files from our host system to the target system.

We use the Virtual Network Computing (VNC) software to provide control and data transfer between the Windows host computer and the CM3. VNC was first developed by Olivetti labs and the base code is open source and currently the company RealVNC maintains the software. The versions we use here are VNC Viewer on the CM3 and Windows. VNC comes preinstalled on the later versions of the CM3 while for other operating systems, it can be installed separately.

We will use the `/tmp` directory on the CM3 for our labs and projects for convenience in system management.

3.1 Transferring files from host to CM3.

1) From the Windows desktop, click on the VNC icon to enter into the VNC environment. You should see the VNC server display showing the CM3 desktop with a Title bar.

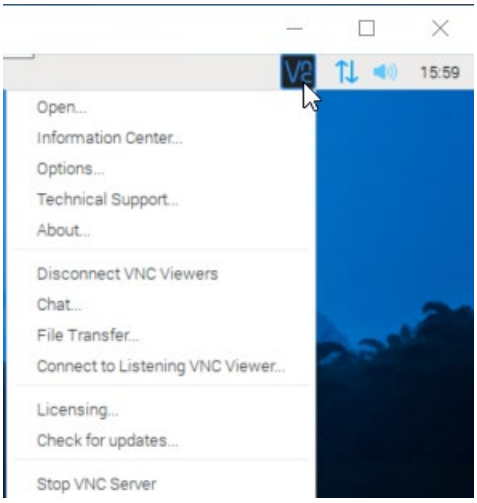
Figure 1 Title bar of VNC server (on target system)

If this is not shown, refer to the Appendix on how to sign into the CM3 using VNC and have the title bar displayed.

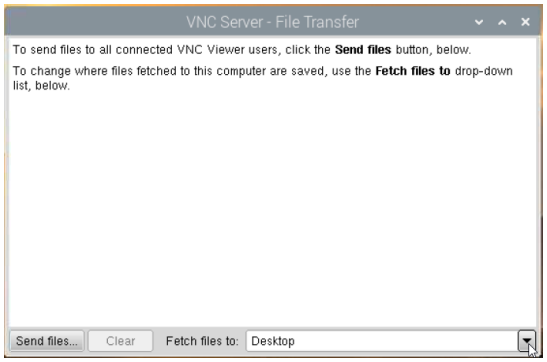
2) Set default transfer directory on CM3.

i. ***Right click*** on the dark VNC icon and a drop down menu appears.

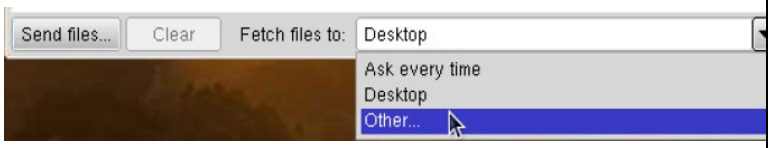
ii. Click on ***File Transfer*** and a file selection dialog will appear.



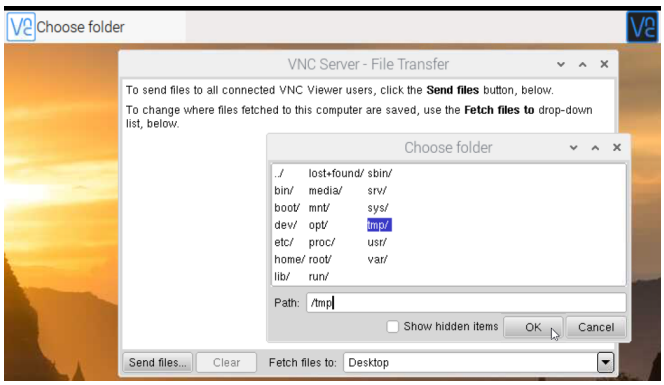
iii. Click on the drop down button at the lower right combo-box ***Fetch files to:***



iv. In the combo-box, select ***Other…*** to open up a dialog.



v. In the directory selection dialog - in the ***Path:*** prompt, type in ***/tmp*** and click OK to set the path.



vi. After this, ***close*** the File Transfer dialog box.

Now all future file transfers from the host will go to the `/tmp` directory by default.

2) To transfer our image files to the CM3, hover the mouse over the ***middle*** of Title Bar of the VNC Viewer. A drop-down menu will appear. Click on the ***Transfer files*** icon and then select ***Send files*** at the lower left of the screen.

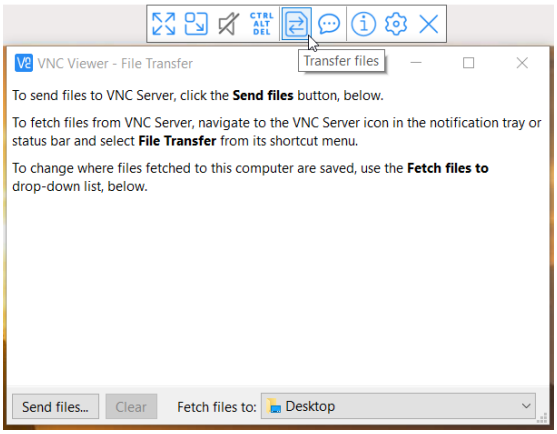
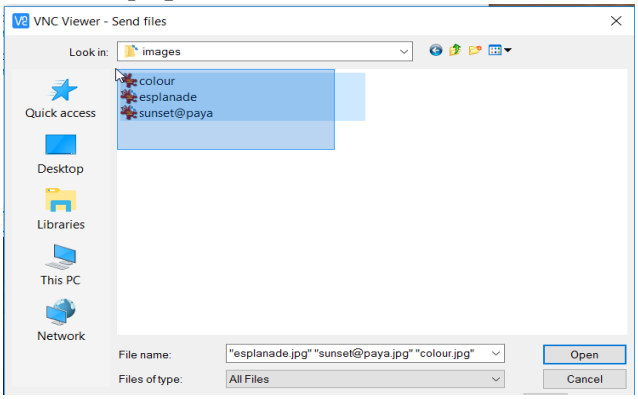
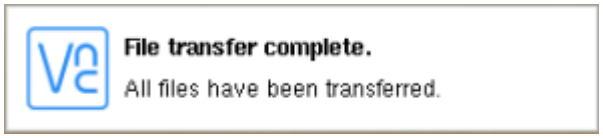


Figure 2 Hover over the middle of Title Bar to activate drop down menu

3) A Windows file selection dialog will open and you will select three images in `D:\ECSLAB\images`. They are the jpeg files : `colour`, `esplanade` and `sunset@paya`.



5) Confirm the selection by clicking ***Open***. VNC will immediately transfer the files to the default directory on the CM3. A confirmatory message box will appear after that.



4) ***Close*** the VNC file transfer dialog.

3.2 Displaying the image files on the CM3

Now that the image files have been transferred over, we will display them under program control. There are two parts to this step which needs some consideration because VS Code and VNC will be sharing the *same* display. Make sure the screens from both programs *do not* obstruct each other totally. First is to establish a connection between the target and host and second is to start a program which will display the images.

i) VNC: In the VNC title bar (section 3.1) click on the terminal icon  to open up a terminal between the target and host system.
DO NOT MAXIMIZE THE SCREEN!

ii) VS Code: Start the VS Code application and open the directory `D:\ECSLAB\lab7`.
DO NOT MAXIMIZE THE SCREEN!
Build and execute the program `lab7.c`.

MAKE SURE YOU ARE ABLE TO VIEW PORTIONS OF BOTH SCREENS!

iii) Follow the prompts on the LCD to display 3 images. Resize and move the screens of VNC and VS Code as needed.

Now we will modify the program to display one more image that we will prepare to show on the CM3.

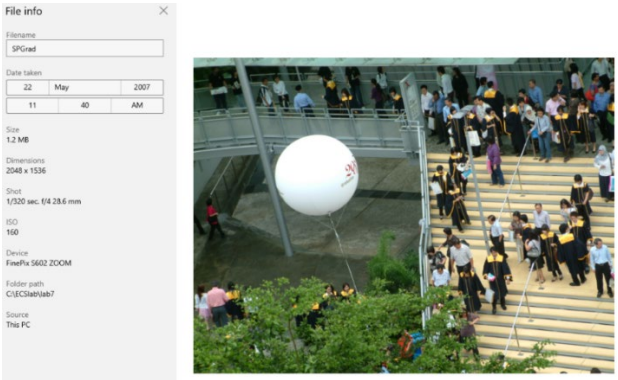
4. Software Used

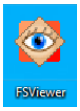
We used an image editing software known as FastStone which is available as free ware, with thanks to the author. This software allows you to resize and change the colour depth of your images. It can also add annotations like text and simple graphics to your image.

To acquaint ourselves with the capabilities of this software, we will prepare an image for use on the embedded system. It is `SPGrad.JPG` and can be found in: `D:\ECSlab\images` directory.

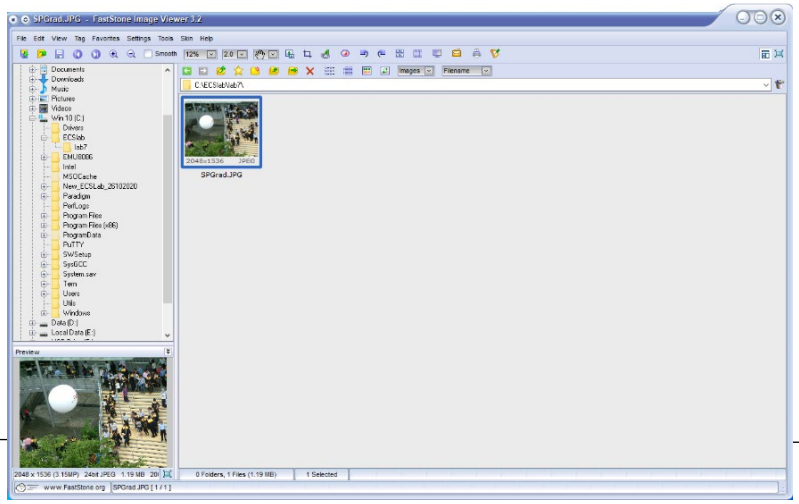
1) The colour depth is 24-bits, by using an image viewer to check the properties of `SPGrad.JPG`, what is the resolution of the image?

_____



2) Start the FastStone program by double-clicking on the icon. 

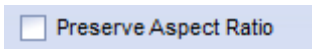
After the application loads, the main display will show. Note that the directory of the program may not be exactly the same as what is shown, as the software will remember the last directory it worked with.

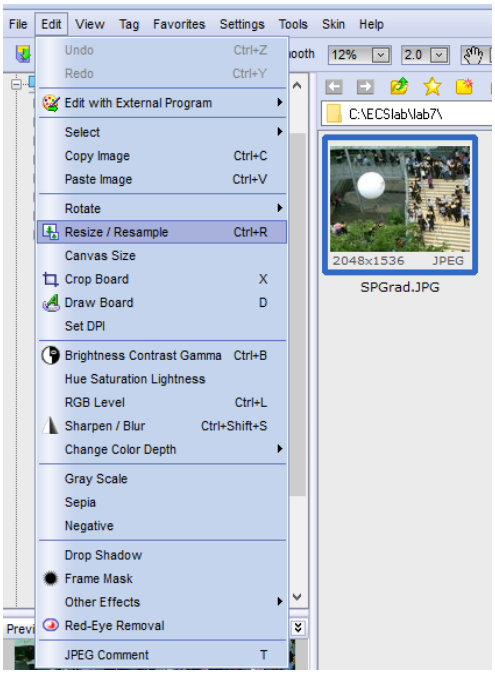


4.1 Resizing the image

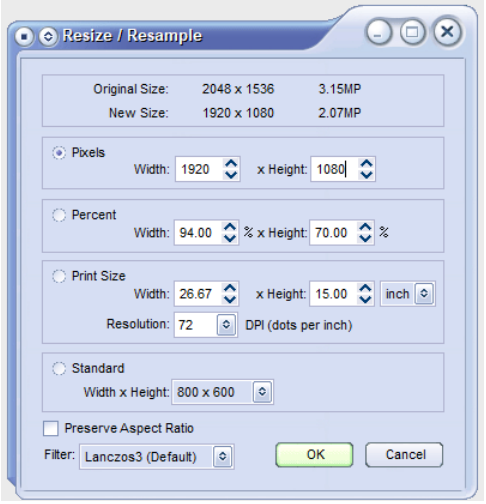
3) To illustrate the issues at hand, we will first resize an image right away.
   The supplied image, named `SPGrad.JPG` needs to be 1920x1080 pixels with 24-bit colour depth, or 16.7 million Colours.
4) Click on `SPGrad.JPG`, make sure the box outside the image is blue.
   Click on the edit option on the top left-hand corner of the application.
5) Then select "Resize / Resample"

   Before changing the aspect ratio, ensure that the "Preserve Aspect Ratio" is unchecked at the bottom of the window.



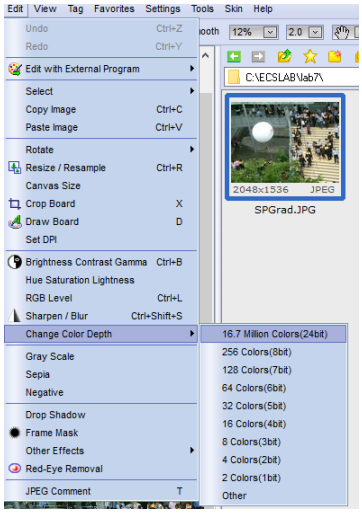


With pixels selected, change the Width to 1920 and Height to 1080 pixels and ***click*** OK.

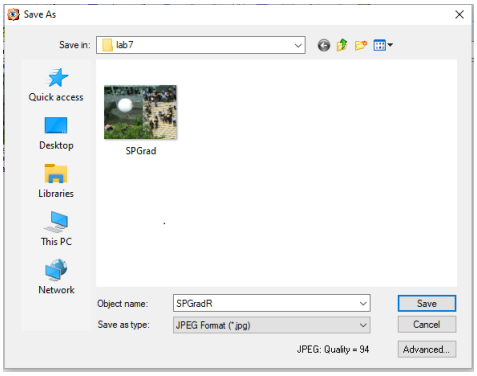


Note: The shortcut for this function is Ctrl+R

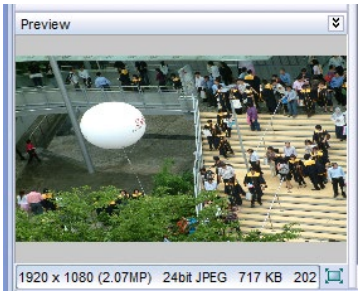
6) Double check that the colour depth is 24bits or 16.7 million Colours by selecting “Change Color Depth” under “Edit” tab.
Make sure “16.7 Million Colors(24bit)” is highlighted.



7) Begin to save the image.
Click on File and select “Save As”
Name the image as SPGradR.jpg in JPEG Format.



The image saved will be shown in the preview window on the bottom left-hand of your window screen.



4.2 Aspect Ratio

The image that has been resized should fit the entire monitor screen as the aspect ratio of the original image is 1.33 (2048 / 1536, 4:3) while that of the new resized image is 1.77 (1920 / 1080, 16:9)

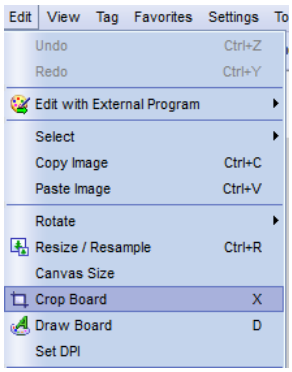
	
SPGrad.jpg (2048 x 1536 / 4:3)	SPGradR.jpg (1920 x 1080 / 16:9)

This resizing step that you have just done is to illustrate the guidelines one has to follow in order to properly prepare an image for display in an LCD screen used by the SBC.
However, in some cases, the distorted image is tolerable.

4.3 Cropping an Image

In order to maintain a desired aspect ratio for resizing, one has cut off a part of the original image to achieve the aspect ratio of 1.6 (16:10). This is known as “crop”.
Before doing so, we should do some calculations. Say we want to keep the image width, then the new dimensions for the desired will have to be 2048 / 1280 (16:10).

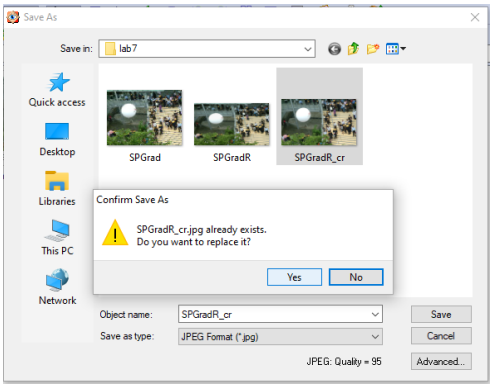
1) Click on SPGrad.JPG, make sure the box highlight outside the image is blue.
   Click on Edit > Crop Board



2) Using the Crop Board tool, Enter "2048 x 1280" as the dimensions at the bottom left-hand corner of the window.

3) The dotted box shown can be moved to indicate which part of the picture to retain.
The dark area outside the dotted box is the part to be cropped away. Afterwards, click Loseless Crop to File > Save it as SPGradCR.jpg

4) Select SPGradCR.jpg, make sure the highlight box is blue around the image, and resize it to 1920x1080. Double-check that the colour depth is 24-bits.
Save the file under the same name as before, a window will pop out saying "Do you want to replace it?"
Select ***Yes***.



You should now see SPGradCR.jpg is 1920 x 1080 and JPEG on the preview window.

The new images now being proportionate and having retained aspect ratio, double-click onto any of the images in FastStone to view them in full screen. The name of the image with the aspect ratio and file size can be seen on the top left-hand corner. To exit, double-click the image again or press ***Esc***. Press Left or Right arrow keys to viewother images while still in full screen.

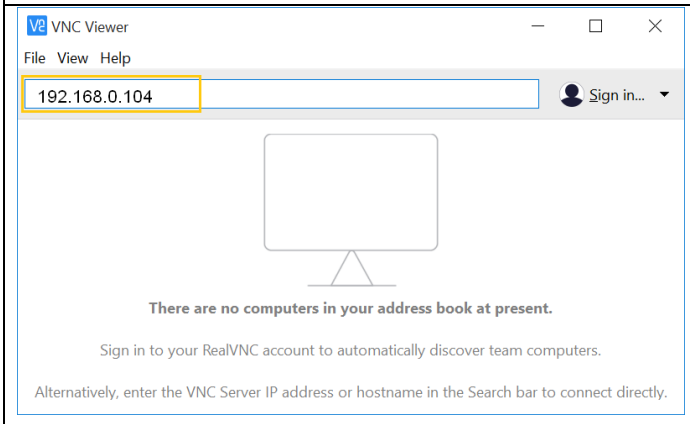
Original – SPGrad.jpg	
Resized, aspect ratio not retained – SPGradR.jpg	
Cropped to aspect ratio, then resized – SPGradRCR.jpg	

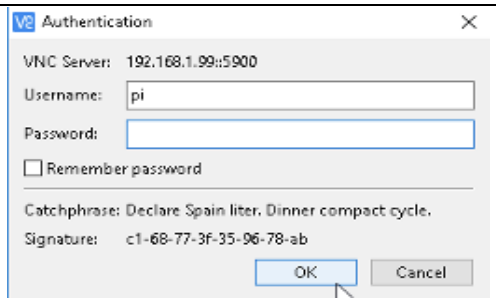
5 Displaying the Edited Image

Now that this image has been correctly resized, transfer it to the CM3 in the directory `/tmp`. Modify lab7.c so it now displays the image together with the rest.

APPENDIX

To log into the CM3 from the development system using VNC, follow these prompts:





Password: ***raspberry***

Introduction to pqiv

In the CM3, we use the program `pqiv` instead of the standard Photo Viewer software because we can call it from a user program in the target system as seen in lab7.c. If pqiv is not available, you may need to install it. The Powerful Quick Image Viewer (pqiv) is written by Phillip Berndt.

However, pqiv can also run from the command line and if you are able to access a Terminal screen connected to a CM3 or Raspberry Pi, you can test its functionality.

If you able to run pqiv when the current directory is `/tmp` you will see the images you uploaded there. The software support various file formats like .bmp, .jpg, .png, .gif but do keep in mind, the bigger the file size, the longer it will take to load.

Using pqiv in a C program

An example of calling pqiv to display the image SPGrad.png in the `/tmp` directory is

```
system("DISPLAY=:0.0 pqiv -f -s --slideshow-interval=1 /tmp/SPGrad.jpg
/tmp/SPGradR.jpg /tmp/SPGradR.jpg &");
```

Here the `system` function will execute the command described in the string and returns from it.

— End of Lab —

Laboratory 8 – Graphics User Interface (GUI)

1. Introduction

The purpose of the laboratory is to experience the design concept of using GUI commonly found in everyday applications. In this laboratory, we will be going through the process in designing the different GUI screens needed for a particular application.

In an earlier lab, we have seen how to edit images, transfer to a target and displaying the images using a program. In this lab, we will add annotations to an image which will be used in a graphics display. These annotated images can be used as part of a user interface.

However in this lab, we will *not* be using graphical control elements that come with a standard graphics library. These control elements, also called widgets – like buttons, edit boxes and so on, require a good understanding of how they interact with an application.

For this lab, we will illustrate how a GUI works by implementing a simple **Stored Value Card Top-Up System.**

2. Objectives

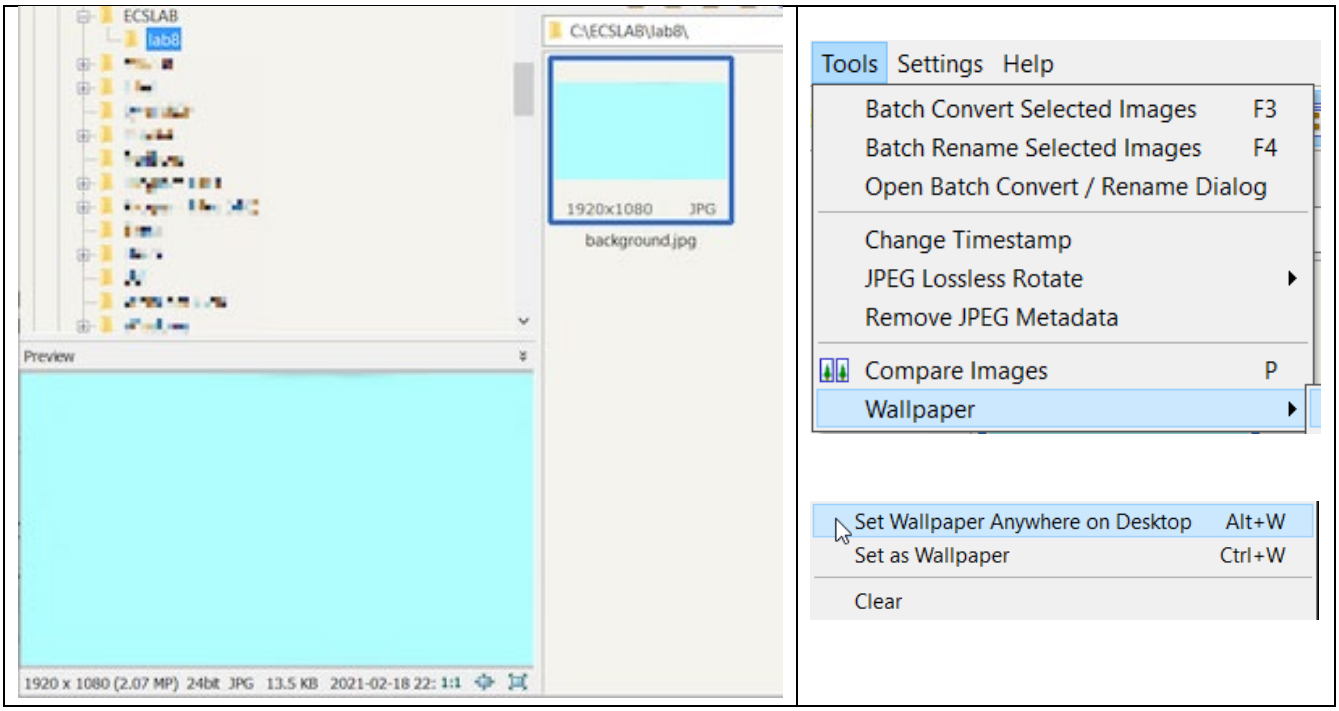
- To gain familiarity with some concepts used in GUI design.
- To gain experience with the GUI display and hardware interaction.

3. Procedure

We will use FastStone to develop the images to be used in the GUI.

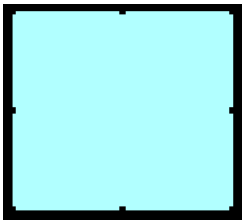
First, we create a background image to act as the base for our other screens.
In the `D:\ECSLAB\images` folder, there is a file named `background.jpg`.

1) Start FastStone, highlight `background.jpg`, click on ***Tools / Set Wallpaper Anywhere on Desktop.***



(Note: File directory listings may differ)

2) Expand the image by clicking the resize button on the side of the image until it covers the window.

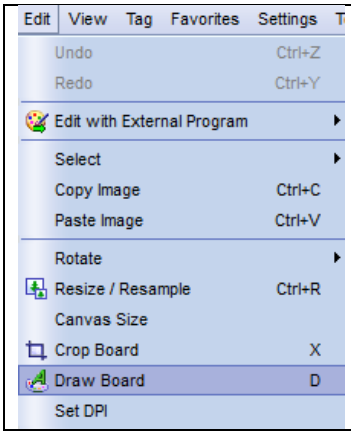


3) Select Mask No.16 on the left-hand side of the screen. You may select any other border as desired.

Click on "Save as File" and name it `bg1.png`. Then press Esc to return to the main display.

2.1 Adding other graphic objects to our image

1) We will now use `bg1.png` as our base screen and modify from there. Select `bg1.png` and edit it by going to the main menu and select ***Edit / Draw Board***. To add text, click on the ***Text*** tool. Click and drag an area on the picture where the text is to be placed.
   Uncheck Background, set the font to ***Arial*** at ***150*** points with a ***red*** colour.



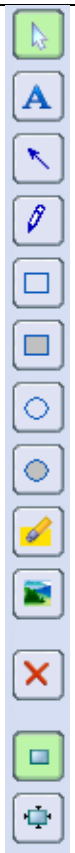
4. After inserting the "Welcome" text, press ***Enter*** to create a new line.
   Change the font size to 100 points and the colour to black.
   Add in more text objects until it looks like `welcome.png` as shown below.

   You can edit the text object you have just inserted by double clicking on the object.

Figure 1 welcome.png

5. When you are done, press ***Esc*** and click ***Ok***. Then save the image as as `welcome.png`. The software can be used to create special effects as we will show later.

Other Draw Tools

There are several drawing tools in the ***Edit / Draw*** option. Explore their capabilities and make of them in your other screens.		Select Object Text Straight Line Pencil Rectangle (Outline) Rectangle (Opaque) Eclipse (Outline) Eclipse (Opaque) Highlight Watermark Image Delete Object Zoom actual size Fit to screen

In all, create four more screens as shown in the next pages and incorporate them in the C program to implement a **Store Value Card Top-Up System**. The next few screens are shown as a guide – but you are encouraged to be creative.

Welcome.png

Enhancing the Welcome text with drop shadows

Assume you have already created a welcome screen in red earlier on. Text can be enhanced by using drop shadows. These appear to give an object a 3D look by inserting a shadow, replicating the object in a darker colour and offset from the object.

1) Select Text to create a new text box and type "Welcome" with Arial font, 150 points in ***black*** this time.
2) Drag the text boxes and position them so the black text is behind the red.

Enhancing the menu choices

1) Select Text and type "2" at 100 pixels in black. Drag the text and position it.
2) Make a new "2" text again but in red. Drag the text and place on top of the black "2".
3) Select Text and type "Press" at 40 pixels in black. Drag the text and position it next to the red "2".
4) Make a new "Press" text but in red. Drag the text and place it on top of the black "Press".

50.png

20.png

Amount.png

ThankYou.png

3. Testing the GUI in the embedded system

1) Transfer the images that were created to the embedded system into the `/tmp` directory.

2) Modify `lab8.c` to correspond to the images you have created.

Laboratory 9 – Multitasking

1. Objectives

- To create separate threads in addition to the main application thread.
- To use mutex for threads synchronization.

In this lab, we illustrate multitasking and the use of synchronizing objects by the use of an application running a Stepper Motor task and a LCD Update task.

2. Introduction

Multitasking is a method by which multiple tasks, also known as process and threads share common processing resources such as a CPU. In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch. When context switches occur frequently enough the illusion of parallelism is achieved.

Tasks can interact with each other directly with each other or through the shared resources. These interactions must be coordinated or synchronized. Event objects are typically used for synchronizing tasks processing rather than for controlling access to shared resources. It is a mechanism that lets a task go to sleep until, for example, some data is ready for it to process or a request is ready for it to service.

When a program runs, the main application thread will be created. From the main application thread more threads can be created as necessary. To create additional threads, the following function can be used :

```
#include <pthread.h>

int main(int argc, char *argv[])
{
    CM3DeviceInit();
    pthread_t thread1, thread2;
    int ptr1, ptr2;
```

```
    ptr2 = pthread_create(&thread1,NULL,thread_motor,NULL);
    ptr1 = pthread_create(&thread1,NULL,thread_LCD,NULL);
}
```

Subsequently the following functions will ensure each thread will complete before terminating :

```
#include <pthread.h>

int main(int argc, char *argv[])
{
    CM3DeviceInit();
    pthread_t thread1, thread2;
    int ptr1, ptr2;

    ptr2 = pthread_create(&thread1,NULL,thread_motor,NULL);
    ptr1 = pthread_create(&thread1,NULL,thread_LCD,NULL);


    pthread_join(thread1,(void**)&ptr;

    pthread_join(thread2,(void**)&ptr;
}
```

Events can be synchronised with one another by using the following functions :

Declarations :

```
pthread_cond_t signalMotor = PTHREAD_COND_INITIALIZER;
pthread_mutex_t lockMotor = PTHREAD_MUTEX_INITIALIZER;
```

Lock motor thread and wait for signal from LCD thread :

```
pthread_mutex_lock(&lockMotor);
pthread_cond_wait(&signalMotor, &lockMotor);
```

Send signal for motor to continue :

```
pthread_cond_signal(&signalMotor);
```

Unlock motor thread :

```
pthread_mutex_unlock(&lockMotor);
```

Other functions can be suspended while a specific function is running by using the following :

Declaration :

```
pthread_mutex_t buslock = PTHREAD_MUTEX_INITIALIZER;
```

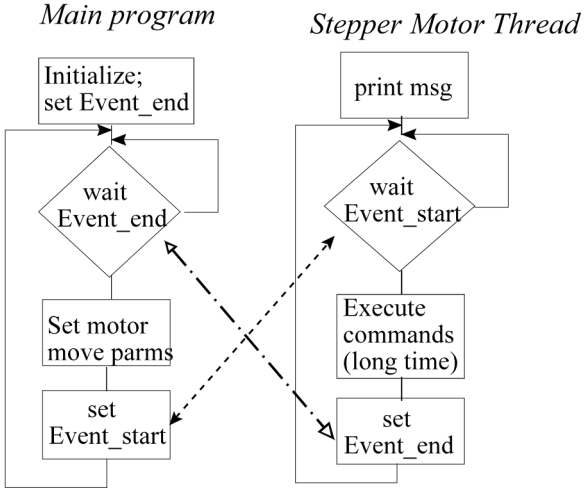
Suspend/Prioritise specific function :

```
pthread_mutex_lock(&buslock);
/* FUNCTION HERE; */
pthread_mutex_unlock(&buslock);
```

We will suspend `CM3_outport()` every time it is called to allow one function to access the output bus at a time. This will ensure the data output from the motor thread and LCD thread do not clash.

3. Functions vs Threads

To appreciate this program, it is good to recall that in a normal C program, the main program calls a function to execute a task. This function will perform its task, terminate and return to the main program which *waits for it* to complete.



However, in a multithreading program, we *begin* a thread which executes *concurrently* with the main program. So there must be some way for the thread to inform the main program when it has finished. It is also time consuming to terminate a thread and start it again. So a thread, after completing its task, should enter a ready state awaiting an event so it is more responsive. Of course, the main program will also be in a ready state. We will illustrate this in the lab.

Briefly, the stepper motor task performs 3 types of moves depending on the integer in `cmd_count` where it counts down sequentially after the thread is completed :

```
void* thread_motor(void* value)
{
    int j;
    int i;
    int steps = 50;
    int cmd_count = 3;

    while(1)
    {
        if (cmd_count == 3)
        {
            LED_func(cmd_count);
            motor_func(steps, CW, motor_fast);
            cmd_count --;
        }
        if (cmd_count == 2)
        {
            LED_func(cmd_count);
            motor_func(steps, CCW, motor_slower);
            cmd_count --;
        }
        if (cmd_count == 1)
        {
            LED_func(cmd_count);
            motor_func(steps, CW, motor_slowest);
            cmd_count = 3;
        }
    }
}
```

This task will execute, terminate and return to the main program where the motor task is called again, perhaps with different parameters.

Meanwhile the LCD is updated with a display of left and right angle brackets. This task runs continuously and so it does not need to be synchronized.

4. Experiment

From the C:\ECSLAB\LAB9 folder, open the lab9_multithread folder. This workspace consists of the source file (lab9.c), a library file (library.c) and one header file (library.h). Fill in the blanks in the program lab9.c and compile the program. When the program has compiled successfully, build the program and run it. Take note of the multitasking aspect of the program.

Modify the program to add one more thread to display the number of steps remaining for each stepper motor command, on the first row of the LCD display. Note that you need to use global variables to access data across threads.

ENGINEERING @ SP

The School of Electrical & Electronic Engineering at Singapore Polytechnic offers the following full-time courses.

1. Diploma in Aerospace Electronics (DASE)

The Diploma in Aerospace Electronics course aims to equip students with skills and competencies in aerospace engineering avionics domains and emerging technologies in Infocomm-Technology to develop them to be a versatile engineer in both aerospace and adjacent industries and also to prepare them for further studies with advanced standing in local and overseas universities.

2. Diploma in Computer Engineering (DCPE)

This diploma aims to train technologists who can design, develop, setup and maintain computer systems; and develop software solutions. Students can choose to specialise in two areas of Computer Engineering & Infocomm Technology, which include Computer Applications, Smart City Technologies (IoT, Data Analytics), Cyber Security, and Cloud Systems.

3. Diploma in Electrical & Electronic Engineering (DEEE)

This diploma offers a full range of modules in the electrical and electronic engineering spectrum. Students can choose one of the six available specialisations (Biomedical, Communication, Microelectronics, Power, Rapid Transit Technology and Robotics & Control) for their final year.

4. Diploma in Engineering with Business (DEB)

Diploma in Engineering with Business provides students with the requisite knowledge and skills in engineering principles, technologies, and business fundamentals, supported by a strong grounding in mathematics and communication skills, which is greatly valued in the rapidly changing industrial and commercial environment.

5. Common Engineering Program (DCEP)

In Common Engineering Program, students will get a flavour of electrical, electronic and mechanical engineering in the first semester of their study. They will then choose one of the 7 engineering courses specially selected from the Schools of Electrical & Electronic Engineering and Mechanical & Aeronautical Engineering.

School of Electrical & Electronic Engineering

More than **65** Years of solid foundation	**8** Tech Hubs	Unique **PTN** Scheme	SP-NUS SP-SUTD Programmes	More than **50,000+** Alumni

SP Singapore Polytechnic

Electives offered by SP SCHOOL OF Electrical & Electronic Engineering

All SP students, including EEE students are free to choose electives offered by ANY SP schools, subject to meeting the eligibility criteria.

Like all schools, School of Electrical and Electronic Engineering offers electives for:

- EEE students only
- all SP students

EEE students are required to complete 3 electives, starting from Year 2 to Year 3 (one elective per semester).

They may also choose to complete 4 or more electives in a particular category to receive a "Minor".

Electives Choices for All SP students

Mod Code	Module Title
EP0400	Unmanned Aircraft Flying and Drone Technologies
EP0401	Python Programming for IoT*
EP0402	Fundamentals of IoT*
EP0403	Creating an IoT Project*
EP0404	AWS Academy Cloud Foundations
EP0405	AWS Academy Cloud Architecting
EP0406	Fundamentals of Intelligent Digital Solutions
EP0407	Technology to Business
EP0408	Cybersecurity Essentials
EP0409	Low Code 5G & AIoT*
EP0410	Fundamentals Of Electric Vehicle
EP0411	5G & Next Generation Networks

Certificate in IoT (Internet of Things)

* A certificate in IoT would be awarded if a student completes any 3 of the following modules: EP0401, EP0402, EP0403 and EP0409

Electives Choices for EEE students

Mod Code	Module Title
EM0400	Commercial Pilot Theory
EM0401	Autonomous Electric Vehicle Design
EM0402	Artificial Intelligence for Autonomous Vehicle
EM0403	Autonomous Mobile Robots
EM0404	Smart Sensors and Actuators
EM0405	Digital Manufacturing Technology
EM0406	Linux Essential
EM0407	Advanced Linux
EM0408	Linux System Administration
EM0409	Rapid Transit System
EM0410	Rapid Transit Signalling System
EM0412	Data Analytics
EM0413	Mobile App Development
EM0414	Client-Server App Development
EM0415	Machine Learning & Artificial Intelligence
EM0416	Solar Photovoltaic System Design
EM0418	Integrated Building Energy Management System
EM0419	Digital Solutioning Skills